Generalized Symmetry Breaking Tasks

Armando Castañeda, Damien Imbs, Sergio Rajsbaum, Michel Raynal

Publications Internes de l'IRISA
ISSN : 2102-6327
PI 2007 – September 2013

Generalized Symmetry Breaking Tasks

Armando Castañeda[*] Damien Imbs[**] Sergio Rajsbaum[***] Michel Raynal[****]

Abstract: Processes in a concurrent system need to coordinate using an underlying shared memory or a message-passing system in order to solve agreement tasks such as, for example, consensus or set agreement. However, coordination is often needed to break the symmetry of processes that are initially in the same state, for example, to get exclusive access to a shared resource, to get distinct names, or to elect a leader.

This paper introduces and studies the family of generalized symmetry breaking (GSB) tasks, that includes election, renaming and many other symmetry breaking tasks. Differently from agreement tasks, a GSB task is inputless, in the sense that processes do not propose values; the task only specifies the symmetry breaking requirement, independently of the system initial state (where processes differ only on their identifiers). Among various results characterizing the family of GSB tasks, it is shown that perfect renaming is universal for all GSB tasks.

The paper then studies the power of renaming with respect to k-set agreement. It shows that, in a system of n processes, perfect renaming is strictly stronger than $(n-1)$-set agreement, but not stronger than $(n-2)$-set agreement. Furthermore, $(n+1)$ renaming cannot solve even $(n-1)$-set agreement. As a consequence, there are cases where set agreement and renaming are incomparable when looking at their power to implement each other.

Finally, the paper shows that there is a large family of GSB tasks that are more powerful than $(n-1)$-set agreement. Some of these tasks are equivalent to n-renaming, while others lie strictly between n-renaming and $(n+1)$-renaming. Moreover, none of these GSB tasks can solve $(n-2)$-set agreement. Hence, the GSB tasks have a rich structure and are interesting in their own. The proofs of these results are based on combinatorial topology techniques and new ideas about different notions of non-determinism that can be associated with shared objects. Interestingly, this paper sheds a new light on the relations linking set agreement and symmetry breaking.

Key-words: Agreement, Asynchronous read/write model, Coordination, Concurrent object, Crash failure, Decision task, Distributed computability, Non-determinism, Problem hierarchy, Renaming, Set agreement, Symmetry breaking, Wait-freedom.

L'univers des tâches réparties de cassage généralisé de la symmétrie

Résumé : *Ce rapport présente une étude exhaustive sur les tâches dont le but est de casser de la symmétrie dans un système réparti.*

Mots clés : *Accord réparti, Asynchronisme, Cassage de la symmétrie, tâche répartie.*

---

[*] Department of Computer Science, Technion, Haifa, Israel
[**] Instituto de Matematicas, UNAM, Mexico
[***] Instituto de Matematicas, UNAM, Mexico
[****] Institut Universitaire de France & IRISA (équipe ASAP commune avec l'Université de Rennes 1 et Inria)

1 Introduction

[1] Processes of a distributed system coordinate through a communication medium (shared memory or message-passing subsystem) to solve problems. If no coordination is ever needed in the computation, we then have a set of centralized, independent programs rather than a global distributed computation. Agreement coordination is one of the main issues of distributed computing. As an example, *consensus* [30] is a very strong form of agreement where processes have to agree on the input of some process. It is a fundamental problem, and the cornerstone when one has to implement a replicated state machine (e.g.,[26, 48, 53]).

We are interested here in coordination problems modeled as *tasks* [44, 52]. A task is defined by an input/output relation Δ, where processes start with private input values forming an *input vector* I and, after communication, individually decide on output values forming an *output vector* O, such that $O \in \Delta(I)$. Several specific agreement tasks have been studied in detail, such as consensus and *set agreement* [24]. Indeed, the importance of agreement is such that it has been studied deeply, from a more general perspective, defining families of agreement tasks, such as *loop agreement* [43], *approximate agreement* [28] and *convergence* [42].

Motivation While the theory of agreement tasks is pretty well developed e.g. [40], the same substantial research effort has not yet been devoted to understanding coordination problems where "break symmetry" among the processes that are initially in a similar state is needed. Only specific forms of symmetry breaking have been studied, most notably *mutual exclusion* [27] and *renaming* [7]. It is easy to come up with more natural situations related to symmetry breaking. As a simple example, consider n persons (processes) such that each one is required to participate in exactly one of m distinct committees (process groups). Each committee has predefined lower and upper bounds on the number of its members. The goal is to design a distributed algorithm that allows these persons to choose their committees in spite of asynchrony and failures.

Similarly, some attention has been devoted in the past to understanding the relative power of agreement and symmetry breaking tasks, but very little is known. There are only two results [31, 36] that measure the relative power of renaming and set agreement; even more, these result focus in very specific instances of these families of tasks. Indeed, [36] is the first that compares the computability power of renaming and set agreement: it shows that $(n-1)$-set agreement (in k-set agreement, processes agree on at most k input values) is strictly stronger than $(2n-2)$-renaming, namely, $(n-1)$-set agreement solves $(2n-2)$-renaming but not vice versa. Then, [31] showed that k-set agreement solves $(n+k-1)$-renaming. Certainly, [31] considers the adaptive version of $(n+k-1)$-renaming, however, clearly the result has implications for the non-adaptive version.

The aim of this paper is to develop the understanding of symmetry breaking tasks and their relation with agreement tasks, motivated by [36].

Generalized symmetry breaking tasks In this paper we introduce *generalized symmetry breaking* (GSB), a family of tasks that includes election, renaming, *weak symmetry breaking* [36, 44][2], and many other symmetry breaking tasks. A GSB task for n processes is defined by a set of possible output values, and for each value v, a lower bound and an upper bound (resp., ℓ_v and u_v) on the number of processes that have to decide this value. When these bounds vary from value to value, we say it is an *asymmetric* GSB task. For example, we can define the *election* asymmetric GSB task by requiring that exactly one process outputs 1 and exactly $n-1$ processes output 2 (in this form of election, processes are not required to know which process is the leader).

In the *symmetric* case, we use the notation $\langle n, m, \ell, u\rangle$-GSB to denote the task on n processes, for m possible output values, where each value has to be decided at least ℓ and at most u times. In the m-renaming task, the processes have to decide new distinct names in the set $[1..m]$. Thus, m-renaming is nothing else than the $\langle n, m, 0, 1\rangle$-GSB task. In the *$k$-weak symmetry breaking task* a process has to decide one of two possible values, and each value is decided by at least k and at most $(n-k)$ processes. This is the $\langle n, 2, k, n-k\rangle$-GSB task. Let us notice that 1-WSB is the weak symmetry breaking task.

[1] Preliminary versions of the results presented in this report appeared in the proceedings of the 18th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2011) [45], the proceedings of the 10th Latin American Theoretical INformatics Symposium (LATIN 2012) [23], and the proceedings of the 27th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'13) [22].

[2] This task is called *reduced renaming* in [44].

Symmetry breaking tasks seem more difficult to study than agreement tasks, because in a symmetry breaking task we need to find a solution given an initial situation that looks essentially the same to all processes. For example, lower bound proofs (and algorithms) for renaming are substantially more complex than for set agreement (e.g., [18, 44]). At the same time, if processes are completely identical, it has been known for a long time that symmetry breaking is impossible [6] (even in failure-free models). Thus, as in previous papers, we assume that processes can be identified by initial names, which are taken from some large space of possible identities (but otherwise they are initially identical). In an algorithm that solves a GSB task, the outputs of the processes can depend only on their initial identities and on the interleaving of the execution.

The symmetry of the initial state of a system fundamentally differentiates GSB tasks from agreement tasks. Namely, the specification of a symmetry breaking task is given simply by a set of legal output vectors denoted $\mathcal{O}$ that the processes can produce: in any execution, any of these output vectors can be produced for any input vector I (we stress that an input vector only defines the identities of the processes), i.e., $\forall I$ we have $\Delta(I) = \mathcal{O}$. For example, for the election GSB task, $\mathcal{O}$ consists of all binary output vectors with exactly one entry equal to 1 and $n-1$ equal to 2. In contrast, an agreement task typically needs to relate inputs and outputs, where processes should agree not only on closely related values, but in addition the values agreed upon have to be somehow related to the input values given to the processes.

Contributions In this paper we study the GSB family of tasks in the standard asynchronous wait-free read/write crash prone model of computation. Our main contributions are:

- The introduction of the GSB tasks, and a formal setting to study them. It is shown that several tasks that were previously considered separately actually belong to the same family and can consequently be compared and analyzed within a single conceptual framework. It is shown that properties that were known for specific GSB tasks actually hold for all of them. Moreover, new GSB tasks are introduced that are interesting in themselves, notably the k-*slot* GSB task, the *election* GSB task and the k-*weak symmetry breaking* task. The combinatorial properties of the GSB family of tasks are characterized, identifying when two GSB tasks are actually the same task, giving a unique representation for each one.

- The identification of four non-deterministic properties of concurrent shared objects. These properties are in some sense necessary to have a "fair" measure of the relative power between agreement and symmetry breaking tasks. As we shall see, with the usual assumption that an object solving a task is a "black-box" that may produce any valid output value in every invocation, the power of GSB tasks is too low to solve any read/write unsolvable agreement task. One of these notions was implicitly used in [36] to compare $(n-1)$-set agreement and $(2n-2)$-renaming. Here we formally define these notions, study their properties and show that they induce a solvability hierarchy.

- Perfect renaming (i.e. when the n processes have to rename in the set $[1..n]$) is a universal GSB task. This means that any GSB task can be solved given a solution to perfect renaming. Moreover, perfect renaming is strictly stronger than $(n-1)$-set agreement. Namely, perfect renaming can solve $(n-1)$-set agreement but not vice versa. This result is complemented by showing that perfect renaming cannot solve $(n-2)$-set agreement. Therefore, the most any GSB task can do is $(n-1)$-set agreement, since perfect renaming is universal in GSB.

- A large subfamily of GSB tasks is identified, such that each task in the family is strictly stronger than $(n-1)$-set agreement. The internal structure of these family is interesting in its own: it has a subfamily of tasks that lie between perfect renaming and $(n+1)$-renaming. Namely, each of these tasks is strictly weaker than perfect renaming but strictly stronger than $(n+1)$-renaming. Therefore, GSB is a "dense" family whose computability power cannot be captured by the renaming subfamily of tasks.

- It is shown that k-set agreement cannot solve $(n+k-2)$-renaming, whenever k is power of a prime number. This result complements [31], where it is shown that k-set agreement solves $(n+k-1)$-renaming.

Most of our proofs heavily exploit the non-determinism properties of objects. We see this as a by-product of identifying and formalizing these properties. We believe that understanding them more in the future will lead to more and better possibility and impossibility results. In some of our proofs we combine these operational arguments with combinatorial topology techniques from [19].

Related Work After Dijkstra, who mentioned "symmetry" in his pioneering work on mutual exclusion in 1965 [27], the first paper (to our knowledge) to study symmetry in shared memory systems is [14]. It considers two forms of symmetry, and shows that mutual exclusion is solvable only when the weaker form of symmetry is considered. In [55] we encounter for the first time the idea that, although processes have identifiers, there are many more identifiers than processes. This implies comparison-based algorithms (where the only way to use identities is to compare them). The paper studies the register complexity of solving mutual exclusion and leader election. In contrast, several anonymous models where processes have no identifiers (but where they do have inputs, the opposite of our GSB tasks) have been considered, e.g. [9, 46]. In these models processes do not fail, and yet leader election is not solvable. These papers concentrate then in studying computability and complexity of agreement tasks. In [9] a general form of agreement task function is defined, in which processes have private inputs and processes have to agree on the same output, uniquely defined for each input. A full characterization of the functions that can be computed in this model is presented.

A study comparing the cost of breaking symmetry vs agreement appeared in [29], but again with no failures. It compares the bit complexity cost of agreement vs breaking symmetry in message passing models.

The renaming problem considered in this paper is different from the *adaptive renaming* version, where the size of the output name space depends on the actual number of processes that participate in a given execution, and not on the total number of processes of the system, n. The consensus number of perfect adaptive renaming is known to be 2 [21]. In a system with n processes, where p denotes the number of participating processes, adaptive $(2p - \lceil \frac{p}{n-1} \rceil)$-renaming is equivalent to $(n-1)$-set agreement [38, 51]. In [39] it is shown that, when at most t processes can crash, k-set agreement can be solved from adaptive $(p+k-1)$-renaming with $k = t$.

The weak symmetry breaking task was used in [44] to prove a lower bound on renaming. The task requires processes to decide on a binary value, with the restriction that not all processes in the system decide the same value. Thus, weak symmetry breaking is a GSB task, and its adaptive version, *strong symmetry breaking* is not. The strong symmetry breaking task extends a similar restriction to executions where only a subset of processes participate: in every execution in which less than n processes participate, at least one process decides 0. It is known that strong symmetry breaking is equivalent to $(n-1)$-set agreement and strictly stronger than weak symmetry breaking [21, 36].

In [32] a family of 01-*tasks* generalizing weak symmetry breaking is defined. As with weak symmetry breaking, all processes should never decide the same binary value. In addition, for executions where not all processes participate, a 01-task specifies a sequence of bits, $b_1, \ldots, b_{n-1}$. If only x processes participate, not all should decide b_x. In contrast, a GSB task specifies restrictions in terms only of n-size vectors (and is not limited to binary values). The computability power of the 01-family is between $(n-1)$-set agreement and $(2n-2)$-renaming.

An important characteristic of GSB tasks is that their specification does not involve the number of participating processes. This is related to the "output-independence" feature mentioned above, which is not the case with agreement tasks, such as k-test-and-set, k-set agreement, and k-leader election, that are defined in terms of participating sets and, consequently, are adaptive. The three are shown to be related in [15]. In k-*test-and-set* at least one and at most k participating processes output 1. In k-*leader election* a process decides an identifier of a participating process, and at most k distinct identifiers are decided.

Papers considering mixed forms of agreement and symmetry breaking are, group renaming [2, 3], committee decision problem [37] and musical benches [34].

A hierarchy of sub-consensus tasks has been defined in [33] where a task T belongs to class k if k is the smallest integer such that T can be wait-free solved in an n-process asynchronous read/write system enriched with k-set agreement objects. The structure of the set agreement family of tasks is identified in [25, 41] to be a partial order, and it was shown that k-set agreement, even when $k = 2$, cannot be used to solve consensus among two processes. Also, [43] studies the hierarchy of *loop agreement* tasks, under a restricted implementation notion, and identifies an infinite hierarchy, where some loop agreement tasks are incomparable. Set agreement belongs to loop agreement.

Roadmap The paper is made up of 9 sections. Section 2 introduces the basic computation model and defines notions used in the paper. Section 3 introduces the family of generalized symmetry breaking (GSB) tasks, and Section 4 focuses on its combinatorial properties. Then, Section 5 introduces the definition of tasks solving tasks and several notions of non-determinism. Section 6 is on the solvability of GSB tasks. Section 7 investigates the relation linking renaming and set agreement, and, more generally, Section 8 investigates the relation linking GSB tasks and set agreement. Finally, Section 9 concludes the paper.

2 Model of computation

This paper considers the usual asynchronous, wait-free read/write shared memory model where processes can fail by crashing (see, e.g., [12, 49, 54]). We restate carefully some aspects of this model because we are interested in a *comparison-based* and an *index-independent* (called *anonymous* in [7]) solvability notion that are not as common.

2.1 Asynchronous read/write wait-free model

Processes and communication model The system includes $n > 1$ asynchronous processes (state machines), denoted p_1, ..., p_n. Up to $n-1$ processes can fail by crashing (defined formally below). The processes communicate by reading and writing atomic single-writer/multi-reader (1WnR) registers. Given an array $A[1..n]$ of 1WnR atomic registers, only p_i can write into $A[i]$ while any process can read all entries of A. To simplify the notation in the formal model of this section, we make the following assumptions without loss of generality (they affect efficiency but not computability). The shared memory consists of a single array of 1WnR registers A (although the codes of our algorithms use more than one register, several registers can be simulated using a single one). Also, p_i has access to an operation READ(j) that atomically gets the value in $A[j]$. The process p_i also has access to a WRITE(val) operation, such that when p_i invokes it with a parameter val, this value is written to $A[i]$. It is known that an atomic snapshot operation can be implemented in the asynchronous wait-free read/write model [1]. Thus, without loss of generality, we assume that p_i also has available a SNAPSHOT() operation that atomically gets a snapshot of A.

In subsequent sections, processes are allowed to cooperate through certain shared objects, in addition to registers. Thus, additionally to READ, WRITE and SNAPSHOT operations, processes communicate by invoking the operations such objects provide.

Indexes The subscript i (used in p_i) is called the *index* of p_i. Indexes are used for addressing purposes. Namely, when a process p_i writes a value to A, its index is used to deposit the value in $A[i]$. Also, when p_i reads A, it gets back a vector of n values, where the j-th entry of the vector is associated with p_j. However, for GSB tasks we assume that the processes cannot use indexes for computation; we formalize this restriction below.

Configurations, inputs and outputs A *configuration* of the system consists of the local state of each process and the contents of every atomic register. An *initial configuration* is a configuration in which all processes are in their initial states and each register contains an initial value.

Each process p_i has two specific local variables denoted $input_i$ and $output_i$, respectively. Those are used to solve decision tasks (see below). In an *initial state* of a process p_i, its input is supplied in $input_i$, while its $output_i$ is initialized to a special default value $\perp$. Two initial states of a process differ only in their inputs. Each variable $output_i$ is a write-once variable. A process can only write to it values different from $\perp$, and can write such a value at most once. Hence, as soon as $output_i$ has been written by p_i, its content does not change. A state of p_i with $output_i \neq \perp$ is called an *output state.*

Algorithms, steps, runs and schedules A *step* is performed by a single process, which executes one of its available operations, READ, WRITE, SNAPSHOT or an invocation to a shared object, performs some local computation and then changes its local state. The state machine of a process p_i models a *local algorithm* A_i that determines p_i's next step. A *distributed algorithm* is a collection $\mathcal{A}$ of local algorithms $\mathcal{A}_0, \ldots, \mathcal{A}_n$. The initial local state of p_i is the value in $input_i$. As already explained, when a process p_i reaches an output state, it modifies its local $output_i$ component.

A *run* r is an infinite alternating sequence of configurations and steps $r = C_0\ s_0\ C_1\ \ldots$, where C_0 is an initial configuration and C_{k+1} is the configuration obtained by applying step s_k to configuration C_k.

The *participating* processes in a run are processes that take at least one step in that run. Those that take a finite number of steps are *faulty* (sometimes called *crashed*), the others are *correct* (or *non-faulty*). That is, the correct processes of a run are those that take an infinite number of steps. Moreover, a non-participating process is a faulty process. A participating process can be correct or faulty.

A *schedule* is the sequence of steps of a run, without the values read or written; i.e, it only contains the order in which processes took a step and what each operation was. A *view* of process p_i in run r is the sequence of its local

states in $C_0\ C_1\ \ldots$ Two runs are *indistinguishable* to a set of processes if all processes in this set have the same view in both runs.

Identities Each process p_i has an identity denoted id_i that is kept in $input_i$. In this paper, we assume identities are the only possible input values. An identity is an integer value in $[1..N]$, where $N > n$ (two identities can be compared with $<$, $=$ and $>$). We assume that in every initial configuration of the system, the identities are distinct: $i \neq j \Rightarrow input_i \neq input_j$.

Clearly, a process "knows" n, because when it issues a read operation, it gets back a vector of n values. However, initially it does not know the identity of the other processes. More precisely, every input configuration where identities are distinct and in $[1..N]$ is possible. Thus, processes "know" that no two processes have the same identity.

Index-independent algorithms An algorithm $\mathcal{A}$ is *index-independent* if the following holds for every run r and every permutation $\pi()$ of the process indexes. Let r_π be the run obtained from r by permuting the input values according to $\pi()$ and, for each step, the index i of the process that executes the step is replaced by $\pi(i)$. Then r_π is a run of $\mathcal{A}$.

Thus, the index-independence ensures that $p_{\pi(i)}$ behaves in r_π exactly as p_i behaves in r: it decides the same thing in the same step. Let us observe that, if $output_i = v$ in a run r of an index-independent algorithm, then $output_{\pi(i)} = v$ in run r_π. This formalizes the fact that indexes can only be used as an addressing mechanism: the output of a process does not depend on indexes, it depends only on the inputs (ids) and on the interleaving. That is, all local algorithms are identical.

For example, if in a run r process p_i runs solo with $id_i = x$, there is a permutation $\pi()$ such that in run r_π there is a process p_j that runs solo with $id_j = x$. If the algorithm is index-independent, p_j should behave in r_π exactly as p_i behaves in r: it decides (writes in $output_j$) the same value and this occurs in the very same step.

Comparison-based algorithms Intuitively, an algorithm $\mathcal{A}$ is *comparison-based* if processes use only comparisons ($<, =, >$) on their inputs. More formally, let us consider the ordered inputs $i_1 < i_2 < \cdots < i_n$ of a run r of $\mathcal{A}$ and any other ordered inputs $j_1 < j_2 < \cdots < j_n$. The algorithm $\mathcal{A}$ is comparison-based if the run r' obtained by replacing in r each i_ℓ by j_ℓ, $1 \leq \ell \leq n$ (in the corresponding process), is a run of $\mathcal{A}$. Notice that each process decides the same output in both runs, and at the same step. Moreover, note that a comparison-based algorithm is not necessarily index-independent and an index-independent algorithm is not necessarily comparison-based.

2.2 Tasks

Definition A one-shot decision problem is specified by a *task*, which is a triple $\langle \mathcal{I}, \mathcal{O}, \triangle \rangle$, where $\mathcal{I}$ is a set of n-dimensional input vectors, $\mathcal{O}$ is a set of n-dimensional output vectors, and $\triangle$ is a relation that associates with each $I \in \mathcal{I}$ at least one $O \in \mathcal{O}$. This definition has the following interpretation: $\triangle(I)$ is the set of output vectors in executions where, for each process p_i, $I[i]$ is the input of p_i. We say task $\langle \mathcal{I}, \mathcal{O}, \triangle \rangle$ is *bounded* if $\mathcal{I}$ is finite.

From an operational point of view, a task provides a single operation denoted propose(v) where v is the input parameter (if any) provided by the invoking process. Such an invocation returns to the invoking process a value whose meaning depends on the task. Each process can invoke propose$(\cdot)$ at most once.

Solving a task An algorithm $\mathcal{A}$ *solves* a task T if the following holds: each process p_i starts with an input value (stored in $input_i$) and each non-faulty process eventually decides on an output value by writing it to its write-once register $output_i$. The input vector $I \in \mathcal{I}$ is such that $I[i] = input_i$ and we say "p_i *proposes* $I[i]$" in the considered run. Moreover, the decided vector J is such that (1) $J \in \Delta(I)$, and (2) each p_i decides $J[i] = output_i$. More formally,

Definition 1. *An algorithm* $\mathcal{A}$ solves *a task* $(\mathcal{I}, \mathcal{O}, \Delta)$ *if the following conditions hold in every run* r *with input vector* $I \in \mathcal{I}$ *where at most* $n-1$ *processes fail:*

- *Termination. There is a finite prefix of* r *denoted* $dec\_prefix(r)$ *in which, for every non-faulty process* p_i, $output_i \neq \perp$ *in the last configuration of* $dec\_prefix(r)$.

- *Validity. In every extension of $dec\_prefix(r)$ to a run r' where every process p_j ($1 \leq j \leq n$) is non-faulty (executes an infinite number of steps), the values o_j eventually written into $output_j$, are such that $[o_1, \ldots, o_n] \in \Delta(I)$.*

Examples of tasks The most famous task is the *consensus* problem [30]. Each input vector I defines the values proposed by the processes. An output vector is a vector whose entries all contain the same value. Δ is such that $\Delta(I)$ contains all vectors whose single value is a value of I. The *k-set agreement* task relaxes consensus allowing up to k different values to be decided [24]. Other examples of tasks are *renaming* [7], *weak symmetry breaking* (e.g. [44]), *committee decision* [37], and *k-simultaneous consensus* [4].

The tasks considered in this paper As already mentioned, this paper only considers tasks where $\mathcal{I}$ consists of all the vectors with distinct entries in the set of integers $[1..N]$. That is, the inputs are the identities. Thus our tasks are bounded.

3 The family of generalized symmetry breaking (GSB) tasks

After defining the family of generalized symmetry breaking (GSB) tasks and proving some of basic properties, we present some instances of GSB tasks that are particularly interesting.

3.1 Definition and basic properties

Informally, a *generalized symmetry breaking* (GSB) task for n processes, $\langle n, m, \vec{\ell}, \vec{u}\rangle$-GSB, $\vec{\ell} = [\ell_1, \ldots, \ell_m]$, $\vec{u} = [u_1, \ldots, u_m]$, is defined by the following requirements.

- Termination. Each correct process decides a value.
- Validity. A decided value belongs to $[1..m]$.
- Asymmetric agreement. Each value $v \in [1..m]$ is decided by at least ℓ_v and at most u_v processes.

Let us emphasize that the parameters n, m, $\vec{\ell}$ and $\vec{u}$ of a GSB task are statically defined. This means that the GSB tasks are non-adaptive.

When all lower bounds ℓ_v are equal to some value ℓ, and all upper bounds u_v are equal to some value u, the task is a *symmetric* GSB, and is denoted $\langle n, m, \ell, u\rangle$-GSB, with the corresponding requirement replaced by

- Symmetric agreement. Each value $v \in [1..m]$ is decided by at least ℓ and at most u processes.

To define a task formally, let $\mathcal{I}_N$ be the set of all the n-dimensional vectors with distinct entries in $1, \ldots, N$. Moreover, given a vector V, let $\#_x(V)$ denote the number of entries in V that are equal to x.

Definition 2 (GSB Task)**.** *For m, $\vec{\ell}$ and $\vec{u}$, the $\langle n, m, \vec{\ell}, \vec{u}\rangle$-GSB task is the task $(\mathcal{I}_N, \mathcal{O}, \Delta)$, where $\mathcal{O}$ consists of all vectors O such that $\forall\, v \in [1..m] :\ \ell_v \leq \#_v(O) \leq u_v$, and for each $I \in \mathcal{I}_N$, $\Delta(I) = \mathcal{O}$.*

Note that a symmetric GSB task can have an asymmetric representation, for example, $\langle n, n, 1, 1\rangle$-GSB and $\langle n, n, [0, 1, \ldots, 1], [n, 1, \ldots, 1]\rangle$-GSB denote the same GSB task, i.e., they are synonyms, they have the same sets of input and output vectors and the same relation (more on this in the next section). The following is a formal definition of a symmetric GSB task. Note that, by definition of GSB, for every GSB task $\langle \mathcal{I}, \mathcal{O}, \triangle\rangle$, for every $I \in \mathcal{I}$, $\triangle(I) = \mathcal{O}$.

Definition 3 (Symmetric and Asymmetric GSB Tasks)**.** *Let $\langle \mathcal{I}, \mathcal{O}, \triangle\rangle$ be a GSB task on m decision values. We say $\langle \mathcal{I}, \mathcal{O}, \triangle\rangle$ is* symmetric *if and only if for every $O \in \mathcal{O}$ and every permutation π of $[1, \ldots, m]$, the permuted vector $[\pi(O[1]), \ldots, \pi(O[n])] \in \mathcal{O}$; otherwise $\langle \mathcal{I}, \mathcal{O}, \triangle\rangle$ is* asymmetric.

We say that the GSB task is *feasible* if $\mathcal{O}$ is not empty. The following lemma is easy to prove.

Lemma 1. *A GSB task is feasible if and only if* $\sum_{v=1}^{m} \ell_v \leq n \leq \sum_{v=1}^{m} u_v$.

For the case of symmetric GSB tasks, the previous lemma can be re-stated as follows.

Lemma 2. *If* $\forall\ v \in [1..m] :\ \ell_v = \ell$ *and* $\forall\ v \in [1..m] :\ u_v = u$*, then the GSB task is feasible if and only if* $m \times \ell \leq n \leq m \times u$.

We fix for this paper $N = 2n - 1$. Thus, all the GSB tasks considered have the same set of input vectors, $\mathcal{I}_{2n-1}$, denoted henceforth simply as $\mathcal{I}$. The following lemma says that considering a set of identities of size larger than $2n-1$ is useless. A similar result is known for renaming (e.g., [17]).

Theorem 1. *Consider two* $\langle n, m, \vec{\ell}, \vec{u}\rangle$*-GSB tasks,* $(\mathcal{I}_N, \mathcal{O}, \Delta)$*,* $N \geq 2n-1$*, and* $(\mathcal{I}, \mathcal{O}, \Delta)$ *(whose only difference is in the set of input vectors). Then* $(\mathcal{I}_N, \mathcal{O}, \Delta)$ *is wait-free solvable if and only if* $(\mathcal{I}, \mathcal{O}, \Delta)$ *is wait-free solvable.*

Proof. If $(\mathcal{I}_N, \mathcal{O}, \Delta)$ is wait-free solvable so is $(\mathcal{I}, \mathcal{O}, \Delta)$, because $\mathcal{I}$ is a subset of $\mathcal{I}_N$.

Assume that there is a wait-free algorithm $\mathcal{A}$ that solves $(\mathcal{I}, \mathcal{O}, \Delta)$. To solve $(\mathcal{I}_N, \mathcal{O}, \Delta)$, processes get new intermediate identities using any index-independent $(2n-1)$-renaming algorithm, such as the one in [12], running it with their initial identities from $\mathcal{I}_N$. The intermediate identities obtained belong to $\mathcal{I}_{2n-1} = \mathcal{I}$. The processes run $\mathcal{A}$ using these identities, to solve $(\mathcal{I}, \mathcal{O}, \Delta)$. The outputs produced by this algorithm belong to $\mathcal{O}$, and a solution to $(\mathcal{I}_N, \mathcal{O}, \Delta)$ is obtained. $\square_{Lemma\ 1}$

Recall that an algorithm is comparison-based if processes only use comparison operations on their inputs. The following lemma generalizes another known (e.g., [17, 21]) property about renaming and weak symmetry breaking. It states that we can assume without loss of generality that a GSB algorithm is comparison-based. This is useful for proving impossibility results (e.g., [10, 18]).

Theorem 2. *Consider an* $\langle n, m, \vec{\ell}, \vec{u}\rangle$*-GSB task,* $T = (\mathcal{I}, \mathcal{O}, \Delta)$*. There exists a wait-free algorithm for* T *if and only if there exist a comparison-based wait-free algorithm for* T.

Proof. Assume there is a wait-free algorithm $\mathcal{A}$ for T. To get a comparison-based wait-free algorithm for T, processes first obtain new, temporary identities invoking any comparison-based $(2n-1)$-renaming algorithm (such as the ones described in [12, 54]), running it with their initial identities from $\mathcal{I}$. The intermediate identities obtained belong again to $\mathcal{I}_{2n-1} = \mathcal{I}$. But now the processes use these identities to run $\mathcal{A}$, and solve T, and the resulting algorithm is comparison-based. This construction is from Eli Gafni. The other direction holds trivially. $\square_{Lemma\ 2}$

3.2 Instances of generalized symmetry breaking tasks

Election We can define the *election* asymmetric GSB task, by requiring that exactly one process outputs 1 and exactly $n-1$ processes output 2, namely, $\langle n, 2, [1, n-1], [1, n-1]\rangle$-GSB.

The election GSB task looks similar to the Test&Set task where there are two values, 1 and 2, such that 1 is decided by one and only one process while 2 is decided by the other processes. The difference is that Test&Set is adaptive, meaning that in every execution (independently of the number participating processes) one process decides 1, while in the election GSB task this is guaranteed only when all processes decide.

k-Weak symmetry breaking with $k \leq n/2$ (k-WSB) This is the $\langle n, 2, k, n-k\rangle$-GSB task which has a pretty simple formulation. A process has to decide one of two possible values, and each value is decided by at least k and at most $(n-k)$ processes. Let us notice that 1-WSB is the well-known weak symmetry breaking (WSB) task.

m-Renaming In the m-renaming task the processes have to decide new distinct names in the set $[1..m]$. It is easy to see that m-renaming is nothing else than the $\langle n, m, 0, 1\rangle$-GSB task.[3]

[3] If m depends on the number of participating processes, the problem is called *adaptive* m-renaming task which is not a GSB task.

Perfect renaming The *perfect renaming* task is the renaming task instance whose size m of the new name space is "optimal" in the sense that there is no solution with $m' < m$ whatever the system model. This means[4] that $m = n$. It is easy to see that this is the $\langle n, n, 1, 1\rangle$-GSB task.

k-Slot This is a new task, defined as follows. Each process has to decide a value in $[1..k]$ and each value has to be decided at least once. This is the $\langle n, k, 1, n\rangle$-GSB task, or its synonym, the $\langle n, k, 1, n-k+1\rangle$-GSB task. As we can see the 1-WSB task (classical weak symmetry breaking) is nothing else than the 2-slot task.

Section 6 will study the difficulty of solving GSB tasks, their relative power among themselves, and the difficulty of each one of the previous GSB tasks. As we shall see, some GSB tasks are solved trivially (i.e., with no communication at all). As an example, this is the case of m-renaming, $m = 2n-1$, namely the $\langle n, 2n-1, 0, 1\rangle$-GSB task (as processes have identities between 1 and $2n-1$, a process can directly decide its own identity). In contrast, some GSB tasks are not wait-free solvable, such as perfect renaming. We shall see that perfect renaming is universal among GSB tasks.

4 Combinatorial properties of GSB tasks

This section studies the combinatorial structure of symmetric GSB tasks, to analyze the following two issues: synonyms and containment of output vectors. This analysis is not distributed. Distributed complexity and computability issues are addressed in Sections 5-8.

Notice that $G_1 = \langle n, m, \vec{\ell}_1, \vec{u}_1\rangle$-GSB and $G_2 = \langle n, m, \vec{\ell}_2, \vec{u}_2\rangle$-GSB may actually be the same task T (i.e., both have the same set of output vectors). In this case we write $G_1 \equiv G_2$, and say that G_1 and G_2 are *synonyms*. For example, $\langle n, 2, 1, n-1\rangle$-GSB, $\langle n, 2, 0, n-1\rangle$-GSB, and $\langle n, 2, 1, n\rangle$-GSB are synonyms.

Also, if the set $S(T_1)$ of the outputs vectors of a GSB task T_1 is contained in the set $S(T_2)$ of the outputs vectors of a GSB task T_2, then clearly T_2 cannot be more difficult to solve than T_1. As $S(T_1) \subset S(T_2)$, any algorithm solving T_1 also solves T_2. In this case, we write $T_1 \subset T_2$.

4.1 Counting vectors and kernel vectors associated with a task

Let T be an $\langle n, m, \ell, u\rangle$-GSB task defined by the set of output vectors $S(T)$. We associate with T a set of vectors (called *counting vectors* and *kernel vectors*) defined as follows.

Definition 4. *Let $O \in S(T)$. The* counting vector *V associated with O is the m-dimensional vector such that $\forall\, v \in [1..m]$: $V[v] = \#_v(O)$. Let $C(T)$ be the set of counting vectors associated with T.*

It follows from the fact that we consider symmetric agreement, that the counting vectors containing the very same values (e.g., $[a, b, c]$, $[b, c, a]$ and $[c, a, b]$ when considering $m = 3$) can be represented by a single counting vector $K[1..m]$, namely, the single vector in which each entry is greater or equal to the next one (e.g., the counting vector $[b, c, a]$ if $b \geq c \geq a$). Such a vector represents all the output vectors of $S(T)$ in which the most frequent value appears $K[1]$ times, the second most frequent value appears $K[2]$ times, etc.

Definition 5. *Let us partition $C(T)$ into sets X of counting vectors such that each set X contains all the counting vectors that are permutations of each other.*

- *The* kernel vector *of X is its counting vector K such that $K[1] \geq K[2] \geq \cdots \geq K[m]$.*
- *The* kernel set *of T is the set of all its kernel vectors.*
- *The* balanced kernel vector *of T is its kernel vector such that $[\frac{n}{m}, \cdots, \frac{n}{m}]$ if n is a multiple of m, and $K = [\lceil \frac{n}{m} \rceil, \cdots, \lfloor \frac{n}{m} \rfloor]$ (with the first $n \mod m$ entries equal to $\lceil \frac{n}{m} \rceil$) if n is not a multiple of m.*

The next lemma follows directly from the definition of *kernel vector* and *kernel set*.

[4] The new name space is $[1..p]$ for perfect adaptive renaming (where p is the number of participating processes).

Lemma 3. *Given a task T, its kernel set is totally ordered by the (usual) lexicographical ordering.*

Summarizing,

- The set of $\langle n, -, -, - \rangle$ GSB tasks is partially ordered (according to the inclusion relation on kernel sets defining tasks),
- If $T1 \subset T2$, any vector (solution) of $T1$ is a vector (solution) of $T2$ from which we conclude that any algorithm that solves $T1$ also solves $T2$.

Examples All the $\langle n, m, \ell, u \rangle$-GSB tasks that are feasible with $n = 6$, $m = 3$ and $u \leq n = 6$ are described in Table 1. Hence, the 6 processes can decide up to 3 different values. The kernel vectors of each of these tasks is indicated, and these kernel vectors are listed according to their lexicographical order, from left to right.

As an example, the kernel vector $[4, 2, 0]$ represents all the output vectors in which the most frequent value (that is 1, 2 or 3) appears 4 times, the second most frequent value appears twice and the third possible value does not appear. As another example, the kernel set of the $\langle 6, 3, 0, 4 \rangle$-GSB task is made up of five kernel vectors, namely, $\{[4, 2, 0], [4, 1, 1], [3, 3, 0], [3, 2, 1], [2, 2, 2]\}$. Let us finally observe that the balanced kernel vector $[2, 2, 2]$ belongs to all tasks. Moreover, the GSB tasks $\langle 6, 3, 2, 5 \rangle$, $\langle 6, 3, 2, 4 \rangle$, $\langle 6, 3, 2, 3 \rangle$, $\langle 6, 3, 0, 2 \rangle$, $\langle 6, 3, 1, 2 \rangle$ and $\langle 6, 3, 2, 2 \rangle$ are *synonyms*. The GSB tasks $\langle 6, 3, 1, 6 \rangle$, $\langle 6, 3, 1, 5 \rangle$ and $\langle 6, 3, 1, 4 \rangle$ are also synonyms. Differently, while some tasks are "included" in other tasks (e.g., the kernel vectors associated with any task are included in the kernel set of the $\langle 6, 3, 0, 6 \rangle$-GSB task, there are tasks that are not included one in the other (e.g., the $\langle 6, 3, 1, 4 \rangle$-GSB and $\langle 6, 3, 0, 3 \rangle$-GSB tasks).

kernel vector $\rightarrow$ task $\downarrow$	canonical 4-tuple	$[6,0,0]$	$[5,1,0]$	$[4,2,0]$	$[4,1,1]$	$[3,3,0]$	$[3,2,1]$	$[2,2,2]$
$\langle 6,3,0,6 \rangle$	yes	x	x	x	x	x	x	x
$\langle 6,3,1,6 \rangle$					x		x	x
$\langle 6,3,0,5 \rangle$	yes		x	x	x	x	x	x
$\langle 6,3,1,5 \rangle$					x		x	x
$\langle 6,3,2,5 \rangle$								x
$\langle 6,3,0,4 \rangle$	yes			x	x	x	x	x
$\langle 6,3,1,4 \rangle$	yes				x		x	x
$\langle 6,3,2,4 \rangle$								x
$\langle 6,3,0,3 \rangle$	yes					x	x	x
$\langle 6,3,1,3 \rangle$	yes						x	x
$\langle 6,3,2,3 \rangle$								x
$\langle 6,3,0,2 \rangle$								x
$\langle 6,3,1,2 \rangle$								x
$\langle 6,3,2,2 \rangle$	yes							x

Table 1: Kernels of $\langle n, m, \ell, u \rangle$-GSB tasks (with $n = 6$ and $m = 3$)

Remark It is important to notice that, while a set of kernel vectors can be associated with a task, not all sets of kernel vectors define a task. As an example, a simple look at Table 1 shows that the set of kernel vectors $\{[5, 1, 0], [4, 2, 0]\}$ does not define a task.

4.2 The classes of ℓ-anchored, u-anchored and (ℓ, u)-anchored tasks

This section presents subclasses of GSB tasks that provide us with a better insight into their family structure. More precisely, when we look at the tasks described in Table 1, we see that several GSB tasks are actually synonyms. Hence, it is important to have a single representative for all the GSB tasks that define the same task. This is captured by the notions of ℓ-anchored and u-anchored tasks.

Definition 6 (Anchoring)**.** *Let G be an $\langle n, m, \ell, u\rangle$-GSB task, G' be the $\langle n, m, \ell, \min(n, u+1)\rangle$-GSB task and G'' be the $\langle n, m, \max(0, \ell-1), u\rangle$-GSB task. G is* ℓ-anchored *if G and G' are synonyms. G is* u-anchored *if G and G'' are synonyms. G is* (ℓ, u)-anchored *if it is both ℓ-anchored and u-anchored.*

Hence, if G is ℓ-anchored, increasing the upper bound u does not modify the task and, if G is u-anchored, decreasing the lower bound ℓ does not modify the task. Finally, (as we will see) an (ℓ, u)-anchored $\langle n, m, \ell, u\rangle$-GSB task is the hardest of the family of $\langle n, m, -, -\rangle$ GSB tasks.

As an example let us consider the family of $\langle 20, 4, -, -\rangle$-GSB tasks. The reader can easily check that $\langle 20, 4, 4, 8\rangle$ is an ℓ-anchored task, $\langle 20, 4, 2, 6\rangle$ is a u-anchored task, $\langle 20, 4, 5, 5\rangle$ is an (ℓ, u)-anchored task while $\langle 20, 4, 4, 6\rangle$ is neither an ℓ nor a u-anchored task.

It is easy to see that all $\langle n, m, \ell, n\rangle$ (resp., $\langle n, m, 0, u\rangle$) GSB tasks are ℓ-anchored (resp., u-anchored). These tasks are said to be *trivially* anchored.

Canonical representative of a GSB task Given an $\langle n, m, \ell, u\rangle$-GSB ℓ-anchored task, its *canonical representative* is the $\langle n, m, \ell, u'\rangle$-GSB task such that the $\langle n, m, \ell, u'-1\rangle$-GSB task is not ℓ-anchored. A similar definition applies for an u-anchored task. A task that is neither only ℓ-anchored nor only u-anchored, or that is (ℓ, u)-anchored, is its own representative.

As an example, let us look at Table 1. The $\langle 6, 3, 2, 2\rangle$-GSB task, that is an (ℓ, u)-anchored task, is the representative for the six tasks associated with the single kernel vector $[2, 2, 2]$. The $\langle 6, 3, 1, 4\rangle$-GSB task, that is ℓ-anchored, is the representative for three tasks associated with the kernel set $\{[4, 1, 1], [3, 2, 1], [2, 2, 2]\}$. Finally, the $\langle 6, 3, 1, 3\rangle$-GSB task, that is not anchored, is its own representative: it is the only task associated with the kernel set $\{[3, 2, 1], [2, 2, 2]\}$.

When considering Table 1 there are 7 canonical representative tasks. These canonical tasks are represented in Figure 1 where "$A \to B$" means "A strictly includes B". Let us notice that the representative $\langle 6, 3, 1, 3\rangle$-GSB task is not anchored.

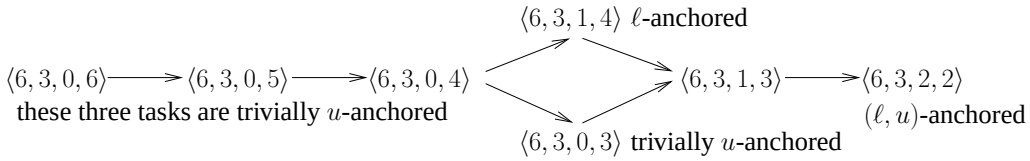

Figure 1: Canonical $\langle n, m, -, -\rangle$ GSB tasks are partially ordered

4.3 A characterization of ℓ-anchored and u-anchored GSB tasks

Let us remember that a task is *feasible* if its set of output vectors $\mathcal{O}$ is not empty.

Theorem 3. *Let T be a feasible $\langle n, m, \ell, u\rangle$-GSB task. T is ℓ-anchored if and only if $u \geq n - \ell(m-1)$.*

Proof. Let us first suppose that $n - \ell(m-1) > u \geq \ell$. As $n - \ell(m-1) \geq u+1$, there is a vector (with m entries) whose first entry is equal to $u+1$ that is a kernel vector of the $\langle n, m, \ell, u+1\rangle$ GSB task. But, as $u+1 > u$, this vector cannot be a kernel vector of the $\langle n, m, \ell, u\rangle$ GSB task. It follows that the $\langle n, m, \ell, u\rangle$ GSB task cannot be ℓ-anchored.

Let us now suppose that $u \geq n - \ell(m-1) \geq \ell$ and consider the counting vector $[n - \ell(m-1), \ell, \ldots, \ell]$ (with m entries). The sum of all its entries is n. Because the occurrence number $n - \ell(m-1)$ is the only value higher than ℓ, it is the highest value that can appear in a kernel vector of both the $\langle n, m, \ell, u\rangle$ task and the $\langle n, m, \ell, u+1\rangle$ for all $u \geq n - \ell(m-1)$. It follows that the $\langle n, m, \ell, u\rangle$ and $\langle n, m, \ell, u+1\rangle$ GSB tasks are the same GSB task from which we conclude that $\langle n, m, \ell, u\rangle$ is ℓ-anchored. $\square_{Theorem\ 3}$

Theorem 4. *Let T be a feasible $\langle n, m, \ell, u\rangle$-GSB task. T is u-anchored if and only if $\ell \leq n - u(m-1)$.*

Proof. The reasoning is similar to the one of Theorem 3. $\square_{Theorem\ 4}$

The next corollary follows from the previous theorems.

Corollary 1. *Let $\ell \leq \frac{n}{m} \leq u$. The $\langle n, m, \ell, \max(\ell, n-\ell(m-1))\rangle$-GSB task is ℓ-anchored, while the $\langle n, m, \max(0, n-u(m-1)), u\rangle$-GSB task is u-anchored.*

4.4 The structural results

Lemma 4. *Let T be any $\langle n, m, \ell, u\rangle$-GSB task. Let $u' \geq u$ and T' be the $\langle n, m, \ell, u'\rangle$-GSB task. We have $S(T) \subseteq S(T')$.*

Proof. The only difference between T and T' is the upper bound on the number of processes that can decide the same value. If at most u processes decide each value, then necessarily less than u' processes decide each value, and thus each output vector of the $\langle n, m, \ell, u\rangle$-GSB task T is also an output vector of the $\langle n, m, \ell, u'\rangle$ task T' and consequently $S(T) \subseteq S(T')$. $\square_{Lemma\ 4}$

Lemma 5. *Let T be any $\langle n, m, \ell, u\rangle$-GSB task. Let $\ell' \leq \ell$ and T' be the $\langle n, m, \ell', u\rangle$-GSB task. We have $S(T) \subseteq S(T')$.*

Proof. The reasoning is similar to the one of Lemma 4. $\square_{Lemma\ 5}$

The next theorem characterizes the hardest task of the sub-family of $\langle n, m, -, -\rangle$-GSB tasks. Let us remember that T_1 is harder than T_2 if $S(T_1) \subset S(T_2)$.

Theorem 5. *The $\langle n, m, \lfloor \frac{n}{m} \rfloor, \lceil \frac{n}{m} \rceil\rangle$-GSB task T is the hardest task of the family of feasible $\langle n, m, -, -\rangle$-GSB tasks.*

Proof. As we consider only feasible tasks, we have $\ell \leq \frac{n}{m} \leq u$. The proof follows then directly from Lemma 4 and Lemma 5. $\square_{Theorem\ 5}$

Let us observe that, given n and m, the $\langle n, m, \lfloor \frac{n}{m} \rfloor, \lceil \frac{n}{m} \rceil\rangle$-GSB task is not necessarily an anchored task. As an example, the $\langle 10, 4, 2, 3\rangle$-GSB task is neither ℓ-anchored nor u-anchored while the $\langle 10, 5, 2, 2\rangle$-GSB task is (ℓ, u)-anchored.

Theorem 6. *Let T be a feasible $\langle n, m, \ell, u\rangle$-GSB task, $T1$ be the $\langle n, m, \ell', u\rangle$-GSB task where $\ell' = n - u(m-1)$ and $T2$ be the $\langle n, m, \ell, u'\rangle$-GSB task where $u' = n - \ell(m-1)$. We have the following: (i) $(\ell' \geq \ell) \Rightarrow S(T1) \subseteq S(T)$ and (ii) $(u' \leq u) \Rightarrow S(T2) \subseteq S(T)$.*

Proof. We prove the theorem for case (i). (The proof for case (ii) is similar.) Let us first show that the $\langle n, m, \ell', u\rangle$-GSB task is feasible, i.e., $\ell' \leq \frac{n}{m} \leq u$. Let us first observe that, as the $\langle n, m, \ell, u\rangle$-GSB task is feasible, by assumption we have $\frac{n}{m} \leq u$. Hence we only have to show that $\ell' \leq \frac{n}{m}$ which is obtained from the following (remember that $m > 1$):

$$\begin{array}{rlcl} & n/m \leq \ u & \Leftrightarrow & n \leq \ u \cdot m \\ \Leftrightarrow & n(m-1) \leq \ u \cdot m(m-1) & \Leftrightarrow & n \cdot m - u \cdot m^2 + u \cdot m \leq \ n \\ \Leftrightarrow & \ell' = n - u \cdot m + u \leq \ n/m. & & \end{array}$$

As $\ell' = n - u(m-1) \leq \frac{n}{m} \leq u$, the size m vector $[u, \ldots, u, \ell']$ is a kernel vector of the feasible $\langle n, m, \ell', u\rangle$ GSB task. As $\ell' \geq \ell$, this vector is also a kernel vector of $\langle n, m, \ell, u\rangle$ GSB task, which concludes the proof for case (i). $\square_{Theorem\ 6}$

The theorem that follows identifies the canonical representative of any feasible $\langle n, m, \ell, u\rangle$-GSB task.

Theorem 7. *Let T be a feasible $\langle n, m, \ell, u\rangle$-GSB task and $f()$ be the function $f(\ell, u) = (\ell', u')$ where $\ell' = \max(\ell, n - u(m-1))$ and $u' = \min(u, n - \ell(m-1))$. The canonical representative of T is the $\langle n, m, \ell_{fp}, u_{fp}\rangle$-GSB task T_{fp} where the pair (ℓ_{fp}, u_{fp}) is the fixed point of $f(\ell, u)$.*

Proof. Let us first observe that, using the same reasoning as in Theorem 6, we have $\ell' \leq \frac{n}{m} \leq u'$, from which follows that T_{fp} is feasible (Lemma 2). Moreover, due to the definition of ℓ' and u', we also have $0 \leq \ell \leq \ell' \leq \frac{n}{m} \leq u' \leq u \leq n$. We consider four cases.

- Case $\ell \geq n - u(m-1)$ and $u \leq n - \ell(m-1)$. We then have trivially $\ell' = \ell$ and $u' = u$, from which we conclude that $S(T)$ and $S(T_{fp})$ have the same kernel vectors.
- Case $\ell' = n - u(m-1) > \ell$ and $u' = u$. Let us consider the kernel vector of T that has as many entries as possible equal to $u = u'$. This means that this vector has $m-1$ entries equal to $u = u'$, and its last entry is equal to $n - u'(m-1)$, i.e., equal to ℓ'. It follows that $S(T)$ has no kernel vector with an entry equal to $\ell'' < \ell'$. We conclude from that observation that the kernel vectors of T are also kernel vectors of T_{fp}, i.e., $S(T) = S(T_{fp})$.
- Case $\ell' = \ell$ and $u' = n - \ell(m-1) < u$. This case is similar to the previous one. Let us consider the kernel vector of T that has as many entries as possible equal to $\ell = \ell'$. This means that this vector has $m-1$ entries equal to $\ell = \ell'$, and its last entry is equal to $n - \ell'(m-1)$, i.e., equal to u'. It follows that $S(T)$ has no kernel vector with an entry equal to $u'' > u'$. Hence, the kernel vectors of T are also kernel vectors of T_{fp}, i.e., $S(T) = S(T_{fp})$.
- Case $\ell' = n - u(m-1) > \ell$ and $u' = n - \ell(m-1) < u$. This case is a simple combination of both previous cases (one addresses the kernel vectors of T with the greatest possible entries, and the other addresses the kernel vectors of T with the smallest possible entries).

According to Theorems 3 and 4, neither the $\langle n, m, \ell'', u\rangle$-GSB task with $\ell'' > \ell'$ nor the $\langle n, m, \ell, u''\rangle$-GSB task with $u'' < u'$ are synonyms of T, which concludes the proof of the Theorem. $\square_{Theorem\ 7}$

5 Tasks solving tasks and non-determinism notions

So far we have defined GSB tasks and studied their internal combinatorial properties. We now focus on solvability issues and compare the computability power of GSB tasks against themselves and against agreement tasks. This section introduces four notions related to non-determinism which are used to compare the computability power between tasks. As the objects (instances of algorithms solving tasks) considered here are assumed to solve a task, they are *one-shot* objects, i.e., each process invokes the object operation at most once. Hence in our model we do not allow processes to locally simulate an object.

5.1 Non-deterministic objects

Let us consider a task $T = \langle \mathcal{I}, \mathcal{O}, \triangle\rangle$ and let $\mathcal{X}$ be an object that solves T.

1. $\mathcal{X}$ is *fully-non-deterministic* (FND) if for every $I \in \mathcal{I}$ and every execution in which processes invoke $\mathcal{X}$ with inputs in I, $\mathcal{X}$ may produce any $O \in \triangle(I)$. Thus, FND agrees with the usual assumption that an object solving a task is a "black-box" that may output any valid output configuration at any time.
2. $\mathcal{X}$ is *unique-solo-deterministic* (USD) if it behaves like an FND object except that there is a unique input value $x \in [1, \ldots, N]$ such that $\mathcal{X}$ is deterministic in all solo-executions (executions in which only one process invokes $\mathcal{X}$) where the input is x, whatever the participating process.
3. $\mathcal{X}$ is *solo-deterministic* (SOD) if it behaves like an FND object except that $\mathcal{X}$ is deterministic in all solo-executions, no matter the participating process and its input.

4. $\mathcal{X}$ is *sequential-deterministic* (SQD) if it behaves like an SOD object that additionally behaves deterministic in every non-concurrent invocation by a single process, namely, the output of $\mathcal{X}$ in a non-concurrent invocation only depends on its internal state (just before the invocation) and the input.

To understand these definitions, consider an object $\mathcal{X}$ that is invoked first by p, then q and r invoke it concurrently (after p's invocation has finished) and finally s invokes $\mathcal{X}$ alone. If $\mathcal{X}$ is FND, then it behaves non-deterministically in every invocation. If $\mathcal{X}$ is USD, then $\mathcal{X}$ behaves non-deterministically in every invocation except p's invocation, only if the input is the unique input for which $\mathcal{X}$ is deterministic in solo-executions, otherwise $\mathcal{X}$ also behaves non-deterministically in p's invocation. If $\mathcal{X}$ is SOD, then it behaves non-deterministically in every invocation except p's invocation. And if $\mathcal{X}$ is SQD, it behaves deterministically in the non-concurrent invocations of p and s, and it behaves non-deterministically in the concurrent invocations of q and r.

5.2 Solvability

Note that an FND object that solves (n, k)-SA is necessarily SOD since there is only one possible output in solo-executions, by the definition of (n, k)-SA. Also observe that any two FND objects solving the same task have the same behavior, in the sense that both may produce the same outputs. However, this is not the case for other objects, SOD objects for example: it is possible that an SOD object outputs y in solo-executions with input x, and another SOD object outputs $z \neq y$ in solo-executions with the same input x. Below we consider algorithms that solve a task from any SOD (resp. USD or SQD) object.

Let T and T' be tasks. For ZZZ $\in$ {FND, USD, SOD, SQD}, we say that T ZZZ-*solves* T', denoted $T \rightarrow_{ZZZ} T'$, if there is a wait-free algorithm $\mathcal{A}$ that solves T' from read/write registers and multiple copies of any ZZZ object $\mathcal{X}$ that solves T. It is required that $\mathcal{A}$ solves T using any ZZZ object $\mathcal{X}$, however, we do not exclude the possibility that processes have an input informing some properties about $\mathcal{X}$ (for example, the outputs the objects may produce in solo-executions). The statement $T \not\rightarrow_{ZZZ} T'$ means $\neg(T \rightarrow_{ZZZ} T')$.

Given two tasks T and T', if there is an algorithm $\mathcal{A}$ that solves T' from FND objects that solves T, then we can obtain an algorithm $\mathcal{B}$ that solves T' by replacing every object solving T in $\mathcal{A}$ with a USD object that solves T. The resulting algorithm $\mathcal{B}$ solves T' because USD objects are indeed FND objects with the property that they behave deterministically in certain solo-executions. Hence the set containing all possible outputs an USD object solving T may produce, is a subset of the set containing all possible outputs an FND object solving T may produce. Thus, if $T \rightarrow_{FND} T'$ then $T \rightarrow_{USD} T'$. Similarly, if $T \rightarrow_{USD} T'$, then $T \rightarrow_{SOD} T'$, and if $T \rightarrow_{SOD} T'$, then $T \rightarrow_{SQD} T'$.

Therefore, the four relations induce a solvability hierarchy: for a GSB task T, let $\mathcal{S}_{ZZZ}$ be the set containing all tasks that T can ZZZ-solve, ZZZ $\in$ {FND, USD, SOD, SQD}; hence $\mathcal{S}_{FND} \subseteq \mathcal{S}_{USD} \subseteq \mathcal{S}_{SOD} \subseteq \mathcal{S}_{SQD}$ (see Figure 2).

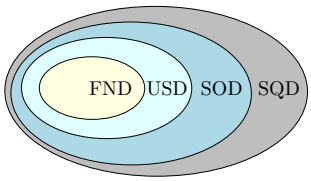

Figure 2: A solvability hierarchy

5.3 On the notions of non-determinism

For proving our computability results in subsequent sections, we consider objects with different non-determinism assumptions, from FND objects to SQD objects. But why do we consider objects holding deterministic properties? Is this an "artificial" way of boosting the computational power of GSB tasks? Arguably, no.

First, the results relating set agreement and renaming in [36] consider different assumptions on the non-deterministic behavior the objects can exhibit. Although it is not explicitly stated, the possibility result that $(n-1)$-set agreement can

implement $(2n-2)$-renaming, assumes $(n-1)$-set agreement objects that are FND. And the impossibility result that, for odd n, $(2n-2)$-renaming cannot solve $(n-1)$-renaming, holds for $(2n-2)$-renaming objects that are SQD (see Appendix B for a detailed explanation). The next theorem shows that the SQD assumption in the impossibility result is reasonable since the computability power of FND objects solving GSB task is null when measured against agreement tasks, or more generally against any read/write unsolvable task without the requirement of index-independent algorithms.

Theorem 8. *Let T be a GSB task and T' be any task which is read/write unsolvable without the requirement of index-independent algorithms. Then, $T \nrightarrow_{FND} T'$.*

Proof. Let $T = \langle \mathcal{I}, \mathcal{O}, \triangle \rangle$. Pick any $O \in \mathcal{O}$. Recall that for every $I \in \mathcal{I}$, $\triangle(I) = \mathcal{O}$. Suppose by contradiction that there is a read/write wait-free algorithm $\mathcal{A}$ that solves T' from FND objects solving T. Let S be the set containing all possible executions of $\mathcal{A}$. Consider the algorithm $\mathcal{B}$ obtained from $\mathcal{A}$ by replacing each object solving T with a local function that always returns $O[i]$, for every process p_i. The fact that $\mathcal{A}$ uses only FND objects implies that every execution of $\mathcal{B}$ belongs to S. Thus, $\mathcal{B}$ must be wait-free and solve T', otherwise $\mathcal{A}$ would not be wait-free nor solve T'. Note that $\mathcal{B}$ is not index-independent, however this is not a problem because solutions to T' do not have to hold that property. Algorithm $\mathcal{B}$ uses only read/write operations, contradicting that T' is read/write unsolvable. $\square_{Theorem\ 8}$

Second, from our perspective, in the task context and without randomness, it is reasonable to assume the non-determinism of shared objects is completely and only due to the possible interleavings of computation steps. Therefore, we believe that the natural way to compare the power of tasks is via SQD objects. For the interested reader, [13] presents a wide discussion about the concept of non-determinism and its relation with concurrency in distributed computing.

Another reason to consider SQD objects is the following. Some tasks have the property that any object that solves one of them is necessarily SQD. This is implied by the very definition of the task. Test&set, for example, has this property. If a process calls alone a test&set object, then the process must get *winner*. In contrast, if two or more processes concurrently call a test&set object, there is no certainty about which process is going to get *winner*. Moreover, in subsequent invocations, a process must get *loser*.

As already explained, for us, the natural way to compare the power of tasks is via the SQD-solvability relation, $\rightarrow_{SQD}$. However, our possibility results hold for FND or USD objects, while our impossibility results hold for SOD or SQD objects.

5.4 Transitivity of FND, USD, SOD and SQD-solvability

FND objects Consider three tasks T_1, T_2 and T_3. Suppose there is an algorithm $\mathcal{A}$ that FND-solves T_2 from T_1. Since T_1 objects in $\mathcal{A}$ are FND, it follows that $\mathcal{A}$ is indeed an FND object that solves T_2. Now, if there is an algorithm $\mathcal{B}$ that FND-solves T_3 from T_2, then we can replace each T_2 object in $\mathcal{B}$ with an instance of $\mathcal{A}$. Therefore, we have that if $T_1 \rightarrow_{FND} T_2$ and $T_2 \rightarrow_{FND} T_3$, then $T_1 \rightarrow_{FND} T_3$.

SOD and SQD objects Similarly, if there is an algorithm $\mathcal{A}$ that SOD-solves T_2 from T_1, then $\mathcal{A}$ is SOD because in every solo-execution the participating process calls SOD objects (which behave deterministically since all calls are indeed solo-executions) and accesses read/write registers without concurrency. Hence $\mathcal{A}$ behaves deterministically in solo-executions. Therefore, if there exists an algorithm that SOD-solves T_3 from T_2, we can replace each T_2 object with an instance of $\mathcal{A}$. We conclude that if $T_1 \rightarrow_{SOD} T_2$ and $T_2 \rightarrow_{SOD} T_3$, then $T_1 \rightarrow_{SOD} T_3$. A similar argument shows the transitivity property of the $\rightarrow_{SQD}$ relation.

USD objects The case of $\rightarrow_{USD}$ is a bit more tricky because it is possible that in an algorithm $\mathcal{A}$ that USD-solves T_1 from T_2, for every input x, in every solo-execution with input x, the participating process invokes some T_1 objects with inputs for which the objects behave non-deterministically, and hence $\mathcal{A}$ may behave non-deterministically in every solo-execution. However, Lemma 6 below shows that given an USD object that solves a GSB task T, using read/write registers and the USD object, one can construct an SOD object that solves T. As explained below, this lemma implies that if T_1 and T_2 are GSB tasks, and $T_1 \rightarrow_{USD} T_2$ and $T_2 \rightarrow_{USD} T_3$, then $T_1 \rightarrow_{USD} T_3$.

Lemma 6. *Let T be any* GSB *task. If there is an* USD *object that solves T, then there is an* SOD *object that solves T.*

Proof.
Let $\mathcal{A}$ be any read/write wait-free comparison-based algorithm that solves $(2n-1)$-renaming, i.e., $\langle n, 2n-1, 0, 1\rangle$-GSB ([21] presents several such algorithms). Consider a process p_i and let E be a solo-execution of $\mathcal{A}$ in which p_i participates. From the fact that $\mathcal{A}$ is comparison-based we get that the output value of p_i in E is not a function of its input value (intuitively, because p_i only uses comparison operations). Thus in every solo-execution, no matter its input, p_i always gets the same output value, say λ. As p_i gets λ in a solo-execution and $\mathcal{A}$ is index-independent, we conclude that in any solo-execution in which p_j participates, $j \neq i$, whatever its input value, p_j gets λ. Let us assume, w.l.o.g., $\lambda = 1$.

Consider now an USD object $\mathcal{X}$ that solves T, and let x be the input such that for every p_i, $\mathcal{X}$ is deterministic in a solo-execution of p_i with input x. Let us assume, w.l.o.g., $x = 1$. Using $\mathcal{A}$ and $\mathcal{X}$, we implement an SOD object that solves T. The idea is to use $\mathcal{A}$ as a preprocessing stage in order that in every solo-execution, the participating process always, invokes $\mathcal{X}$ with input 1, whatever the input. Each process first invokes $\mathcal{A}$ using its original input. Then, a process uses as input to $\mathcal{X}$ the value it gets from $\mathcal{A}$ and finally outputs the value it receives from $\mathcal{X}$. Note that in every solo-execution, the participating process calls $\mathcal{X}$ with input 1, thus the resulting object is SOD because $\mathcal{X}$ is deterministic in solo-executions with input 1. $\square_{Lemma\ 6}$

By Lemma 6, if we have an USD object that solves T_1, we can build an SOD object that solves T_1. And as explained in the previous section, $T_1 \rightarrow_{USD} T_2$ implies $T_1 \rightarrow_{SOD} T_2$. Similarly, Lemma 6 and $T_2 \rightarrow_{USD} T_3$ imply $T_2 \rightarrow_{SOD} T_3$. By transitivity of $\rightarrow_{SOD}$, $T_1 \rightarrow_{SOD} T_3$. Finally, consider an algorithm $\mathcal{A}$ that SOD-solves T_3 from T_1. Lemma 6 implies that given an USD object $\mathcal{X}$ that solves T_1, it is possible to replace each SOD object in $\mathcal{A}$ with a copy of $\mathcal{X}$ and read/write registers in a way that $\mathcal{A}$ stills solves T_3. Therefore, $T_1 \rightarrow_{USD} T_3$.

6 Solvability of GSB tasks

Recall that for a $\langle n, m, \vec{\ell}, \vec{u}\rangle$-GSB task $T = (\mathcal{I}, \mathcal{O}, \Delta)$, we have that $\Delta(I) = \Delta(I') = \mathcal{O}$, for any two input vectors I, I'. Thus, at first sight, it could seem that a trivial solution for T could be to simply pick a predefined output vector $O \in \mathcal{O}$, and always decide it without any communication, whatever the input vector. This is not the case because of the index-independence requirement. In fact, there are GSB tasks that are not wait-free solvable (with *any* amount of communication).

This section investigates the difficulty of solving GSB tasks. In particular, it considers read/write solvable GSB tasks, i.e., for which there exists a wait-free algorithm based only on read/write registers.

As we shall see, the universe of GSB tasks includes trivial tasks that can be solved without accessing the shared memory, and universal tasks, that can be used to solve any other GSB task. In between, there are wait-free solvable tasks, as well as non-wait-free solvable tasks.

6.1 Hardest GSB tasks: Universality of the $\langle n, n, 1, 1\rangle$-GSB task

When considering the GSB family of tasks, an interesting question is the following: is there a universal GSB task? In other words, is there a GSB task that allows all other GSB tasks on n processes to be solved? The answer is "yes". We show in the following that the perfect renaming $\langle n, n, 1, 1\rangle$-GSB task allows any task of the family to be solved. Hence, perfect renaming is *universal* for the family of $\langle n, -, -, -\rangle$-GSB tasks.

As we will see with Corollary 5, the $\langle n, n, 1, 1\rangle$-GSB task (perfect renaming) is not a wait-free solvable task [7]. We present a novel proof of this impossibility result.

Theorem 9. *Any (feasible) $\langle n, m, \vec{\ell}, \vec{u}\rangle$-GSB task can be FND-solved from the perfect renaming $\langle n, n, 1, 1\rangle$-GSB task.*

Proof. Let us first observe that the $\langle n, n, 1, 1\rangle$-GSB task has a single kernel vector, namely, $[1, \ldots, 1]$. Given an algorithm solving that task, let dec_i be the output at process p_i.

To solve the symmetric $\langle n, m, \ell, u\rangle$-GSB task, the processes execute an algorithm solving the $\langle n, n, 1, 1\rangle$-GSB task, and a process p_i considers $output_i = ((dec_i - 1) \mod m) + 1$ as its output. The corresponding kernel vector

for m output values is then $[\lceil \frac{m}{n} \rceil, \ldots, \lceil \frac{m}{n} \rceil, \lfloor \frac{m}{n} \rfloor, \ldots, \lfloor \frac{m}{n} \rfloor]$. By the feasibility assumption, we have $\ell \leq \frac{m}{n} \leq u$. As ℓ and u are integers, we have $\ell \leq \lfloor \frac{m}{n} \rfloor \leq \lceil \frac{m}{n} \rceil \leq u$. The vector $[\lceil \frac{m}{n} \rceil, \ldots, \lceil \frac{m}{n} \rceil, \lfloor \frac{m}{n} \rfloor, \ldots, \lfloor \frac{m}{n} \rfloor]$ is consequently a kernel vector of the $\langle n, m, \ell, u \rangle$-GSB task.

To solve the asymmetric $\langle n, m, \vec{\ell}, \vec{u} \rangle$-GSB task, we first consider the set of output vectors $\mathcal{O}$. We then order these vectors in the same, deterministic way, and pick the first one. Let V be this vector of the $\langle n, m, \vec{\ell}, \vec{u} \rangle$-GSB task. We use then the same vector V for all processes. Let dec_i be the value obtained by process p_i in the $\langle n, n, 1, 1 \rangle$-GSB task. A process p_i then considers $V[dec_i]$ entry as its output $output_i$ with respect to the $\langle n, m, \vec{\ell}, \vec{u} \rangle$-GSB simulated task. Because the $\langle n, n, 1, 1 \rangle$-GSB task has a single kernel vector $[1, \ldots, 1]$, it follows that each entry of V is chosen by only a single process. This satisfies the specification of the $\langle n, m, \vec{\ell}, \vec{u} \rangle$-GSB task, which concludes the proof of the theorem. $\square_{Theorem\ 9}$

6.2 Easiest GSB tasks: Solvability of GSB tasks with no communication

This section identifies the easiest of all the GSB tasks, namely those that are solvable with no communication at all. This is under the assumption that the domain of possible identities is of size $2n-1$ (see Theorem 1). It is easy to see that any feasible GSB task where $m = 1$ is solvable without any communication (a single value can be decided). The next theorem characterizes the communication-free GSB tasks when $m > 1$.

Theorem 10. *Consider an $\langle n, m, \ell, u \rangle$-GSB task T where $m > 1$. Then, T is read/write solvable with no communication if and only if $(\ell = 0) \wedge (\lceil \frac{2n-1}{m} \rceil \leq u)$.*

Proof. Let us first assume $\ell = 0$ and $u = \lceil \frac{2n-1}{m} \rceil$ (increasing u makes the problem even easier). Recall that the identities of the processes are taken from $1..2n-1$. Let us deterministically partition the $2n-1$ identities into m groups, $G_1, \ldots, G_m$, so that no group has more than $\lceil \frac{2n-1}{m} \rceil$ elements and no group has less than $\lfloor \frac{2n-1}{m} \rfloor$ elements. Let δ be the deterministic function that maps identities in group G_i to i(the partitioning and δ are known by every process). To solve T with no communication, each process p_i outputs $\delta(id_i)$ and we have that each value $x \in [1..m]$ is decided by at most $\lceil \frac{2n-1}{m} \rceil$ processes.

For the other direction, let us first consider an $\langle n, m, \ell, u \rangle$-GSB task T with $m > 1$ and $u < \lceil \frac{2n-1}{m} \rceil$. Suppose, by way of contradiction, that there is an algorithm $\mathcal{A}$ that solves T with no communication. The algorithm implies a decision function δ that assigns to each identity x in $1..2n-1$, an output value $\delta(x)$ in $1..m$. The value $\delta(x)$ is the decision produced by a process when it starts with identity x, without any communication. Define groups G_i by putting in the same group identities x, x' whenever $\delta(x) = \delta(x')$. For any partition of the set of identities, the size of the biggest group is at least $\lceil \frac{2n-1}{m} \rceil$. The task specification requires that for each i, $|G_i| \leq u < \lceil \frac{2n-1}{m} \rceil$, which is impossible.

Let us now consider an $\langle n, m, \ell, u \rangle$-GSB task T with $m > 1$ and $\ell > 0$. For any partition of the set of identities, as $m \geq 2$, the size of the smallest group is at most $\lfloor \frac{2n-1}{m} \rfloor \leq n-1$. The task specification requires that, for each i, $|\{p_j \mid \delta(id_j) = i\}| \geq \ell \geq 1$. Because there are $n-1$ identities not corresponding to any process and the size of the smallest group obtained from the partitioning is at most $n-1$, it follows that it is possible that no process belongs to some group, which concludes the proof. $\square_{Theorem\ 10}$

Let us call *x-bounded renaming* the $\langle n, \lceil \frac{2n-1}{x} \rceil, 0, x \rangle$-GSB task. This task can easily be solved, namely, process p_i decides the value $\lceil \frac{id_i}{x} \rceil$.

Corollary 2. *The x-bounded renaming $\langle n, \lceil \frac{2n-1}{x} \rceil, 0, x \rangle$-GSB task is read/write solvable with no communication.*

The next corollary is an immediate consequence of Theorem 10 when $m = 2$ and $\ell = 1$.

Corollary 3. *The* WSB $\langle n, 2, 1, n-1 \rangle$-GSB *task is not read/write solvable without communication.*

When $m = 2n-1$ in Theorem 10, we have the trivial $\langle n, 2n-1, 0, 1 \rangle$-GSB, which is actually the classical (non-adaptive) $(2n-1)$-renaming problem for which many solutions have been proposed (e.g., [5, 8, 15]; see [21]

for an introductory survey). In our setting (where according to Theorem 1, we have $\forall i : id_i \in [1..2n-1]$), to solve $\langle n, 2n-1, 0, 1\rangle$-GSB task each process only has to output its own identity.

Interestingly, as mentioned later, when considering $m = 2n-2$ and the $\langle n, 2n-2, 0, 1\rangle$-GSB task, things become much more interesting. This task may or may not be wait-free solvable, depending on the value of n. The proof of the following corollary is obtained by replacing $(2n-1)$ by $2(n-k)$ in the proof of Theorem 10.

Corollary 4. *The k-WSB $\langle n, 2, k, n-k\rangle$-GSB task is FND-solvable without communication from the $2(n-k)$-renaming $\langle n, 2(n-k), 0, 1\rangle$-GSB task.*

6.3 Hierarchy results, GSB tasks of intermediate difficulty

While the renaming $\langle n, 2n-1, 0, 1\rangle$-GSB task is solvable with no communication, the renaming $\langle n, 2n-2, 0, 1\rangle$-GSB task is not wait-free solvable, except when n is not a prime power [17, 18]. Interestingly, [36] shows that $\langle n, 2n-2, 0, 1\rangle$-GSB and the WSB $\langle n, 2, 1, n-1\rangle$-GSB task are wait-free FND-equivalent: $\langle n, 2, 1, n-1\rangle$-GSB $\rightarrow_{FND}$ $\langle n, 2n-2, 0, 1\rangle$-GSB and $\langle n, 2n-2, 0, 1\rangle$-GSB $\rightarrow_{FND}$ $\langle n, 2, 1, n-1\rangle$-GSB.

Theorem 11. *For every $m > 1$ and $u > 0$, if n is a prime power, then $\langle n, m, 1, u\rangle$-GSB is not read/write solvable.*

Proof. For any $m > 1$, the $\langle n, m, 1, (n-m+1)\rangle$-GSB task solves the WSB $\langle n, 2, 1, n-1\rangle$-GSB task: the processes decide the output of the $\langle n, m, 1, n\rangle$-GSB task modulo 2. It has been shown in [18] that WSB is not read/write wait-free solvable when n is a prime power. The $\langle n, m, 1, (n-m+1)\rangle$-GSB task is then not wait-free solvable either. Moreover, if $m > n$, the $\langle n, m, 1, (n-m+1)\rangle$-GSB task is not feasible. Let us then consider the case in which $n \geq m > 1$. It follows from Theorem 3 that, $\forall m \leq n$, the $\langle n, m, 1, (n-m+1)\rangle$-GSB task is a feasible ℓ-anchored task. Thus, $\forall\ u \geq (n-m+1)$, the $\langle n, m, 1, u\rangle$ and $\langle n, m, 1, (n-m+1)\rangle$-GSB tasks are synonyms. On another side, it follows from Lemma 4 that, $\forall\ n, m, \ell$ and $u' \geq u$, the $\langle n, m, \ell, u\rangle$-GSB task T and the $\langle n, m, \ell, u'\rangle$-GSB task T' are such that $S(T) \subseteq S(T')$. Thus if the $\langle n, m, 1, (n-m+1)\rangle$-GSB task is not wait-free solvable, then the $\langle n, m, 1, u\rangle$-GSB task is not wait-free solvable either for any $u \geq (n-m+1)$, which concludes the proof of the theorem. $\square_{Theorem\ 11}$

Now, consider the election asymmetric GSB task: one process decides 1, while $n-1$ processes decide 2. The output vectors of this task are contained in the output vectors of the WSB $\langle n, 2, 1, n-1\rangle$-GSB task, and hence, election trivially solves WSB. Moreover, election is strictly stronger than WSB because election is not wait-free solvable (see below), while WSB is solvable for (infinitely many) values of n [18].

Theorem 12. *The election $\langle n, 2, [1, n-1], [1, n-1]\rangle$-GSB task is not read/write solvable.*

Proof. Assume for contradiction there is a read/write algorithm $\mathcal{A}$ solving election. By Lemma 2 we can assume $\mathcal{A}$ is comparison based.

Let $\mathcal{B}$ any read/write algorithm in which processes decide either 1 or 2; $\mathcal{B}$ is index-independent and comparison-based. It is known [19] that, for any input configuration I of the system, the number of executions of $\mathcal{B}$ starting from I in which the n processes of the system decide the same value is given by $\#M = 1 + \binom{n}{1}k_1 + \ldots, + \binom{n}{n-1}k_{n-1}$, for some integers $k_1, \ldots, k_{n-1}$. For $b \in \{1, 2\}$, the value of k_i is completely determined by the number of executions with participating processes $p_1, \ldots, p_i$ (each p_j starts with $I[j]$) in which every participating process decides b (if we consider the other decision value instead of b, the k_i change but the expression still holds). If there are no such executions, then $k_i = 0$. (The result in [19] is obtained and expressed using combinatorial topology tools, as in [10, 15, 44, 56]. The operational interpretation we give here is enough for our purposes.)

Since $\mathcal{A}$ solves election, in any execution with participating processes $p_1, \ldots, p_i$, $i \geq 2$, at least one process decides 2. Hence, for $i \geq 2$, there are no executions of $\mathcal{A}$ with participating processes $p_1, \ldots, p_i$ in which all of them decide 1. Consequently, for any input configuration I and setting $b = 1$, the number of executions of $\mathcal{A}$ in which all n processes of the system decide 1 is given by $\#M = 1 + \binom{n}{1}k_1 = 1 + nk_1$, for some integer k_1. Clearly, $\#M \neq 0$, which implies that there is at least one execution of $\mathcal{A}$ in which at least two processes decide 1. A contradiction. $\square_{Theorem\ 12}$

The next corollary follows from the fact that leader election is not read/write solvable and perfect renaming is universal for the family of GSB tasks. This result has been proved in [7], however our proof based in GSB reductions is novel.

Corollary 5. *The perfect renaming* $\langle n, n, 1, 1\rangle$-GSB *task is not read/write solvable.*

7 Renaming vs set agreement

In this section we compare the computability power of the renaming family of GSB tasks with the set agreement family of tasks. The main result of the section is depicted in Figure 3. We first show that the universal GSB task perfect renaming is strictly stronger than the weakest non-trivial $(n, n-1)$-SA. Then we prove that certainly this is the best perfect renaming can do, since it cannot solve $(n, n-2)$-SA, at least from SOD objects. We also show that the renaming $\langle n, n+1, 0, 1\rangle$-GSB cannot solve $(n, n-1)$-SA, again from SOD objects. Hence, in the renaming family of GSB tasks, there is only one member, the strongest one, that is powerful enough to solve a non-trivial set agreement task, which in turn is the weakest member of the set agreement family. In the end of the section we present results that complement upper and lower bounds in [36, 31].

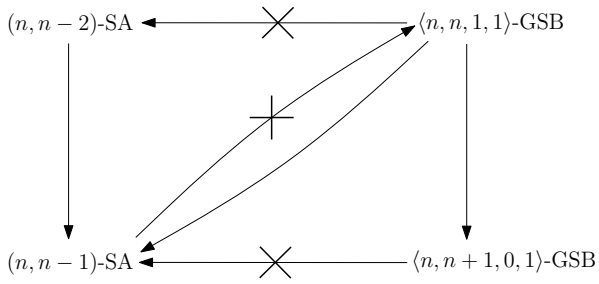

Figure 3: Comparing renaming and set agreement.

Figure 4 depicts the mentioned results and some previous results as well. A solid arrow corresponds to one of our results while a dotted arrow from set agreement to renaming to one in [31]. A cross on an arrow means an impossibility result.

7.1 Perfect renaming is strictly stronger than $(n, n-1)$-SA

First we show that $(n, n-1)$-SA can be done from USD objects solving perfect renaming. Recall that for measuring the computability power of GSB tasks we need to assume some non-determinism properties (see Theorem 8). Second we show $(n, n-1)$-SA cannot solve perfect renaming.

Theorem 13. *For every* n*, the perfect renaming* $\langle n, n, 1, 1\rangle$-GSB *task USD-solves the set agreement* $(n, n-1)$-SA *task.*

Proof.
Let $\mathcal{A}$ be any read/write wait-free comparison-based algorithm that solves the renaming task $\langle n, 2n-1, 0, 1\rangle$-GSB ([21] presents several such algorithms). The fact that $\mathcal{A}$ is comparison-based and index-independent implies there is a value Υ such that for every process p_i, in every solo-execution of p_i, whatever its input name, p_i always gets Υ.

Let $\mathcal{X}$ be an USD object that solves perfect renaming $\langle n, n, 1, 1\rangle$-GSB. By Lemma 6, we can assume $\mathcal{X}$ is SOD. Consider the following object $\mathcal{Y}$ implemented with $\mathcal{A}$ and $\mathcal{X}$: every process calls $\mathcal{A}$ using its original input, and then outputs the value it receives from $\mathcal{X}$, using the value it gets from $\mathcal{A}$ as input to $\mathcal{X}$. Clearly $\mathcal{Y}$ solves $\langle n, n, 1, 1\rangle$-GSB, and as $\mathcal{X}$ is SOD, $\mathcal{Y}$ is SOD. Moreover, note that for every process p_i, in every solo-execution of $\mathcal{Y}$ with participating process p_i, $\mathcal{A}$ outputs Υ to p_i, hence p_i always calls $\mathcal{X}$ with input Υ. Since $\mathcal{X}$ is SOD, it follows that there is a value λ such that for every p_i, in every solo-execution of $\mathcal{Y}$ with participating process p_i, whatever its input name, p_i gets λ.

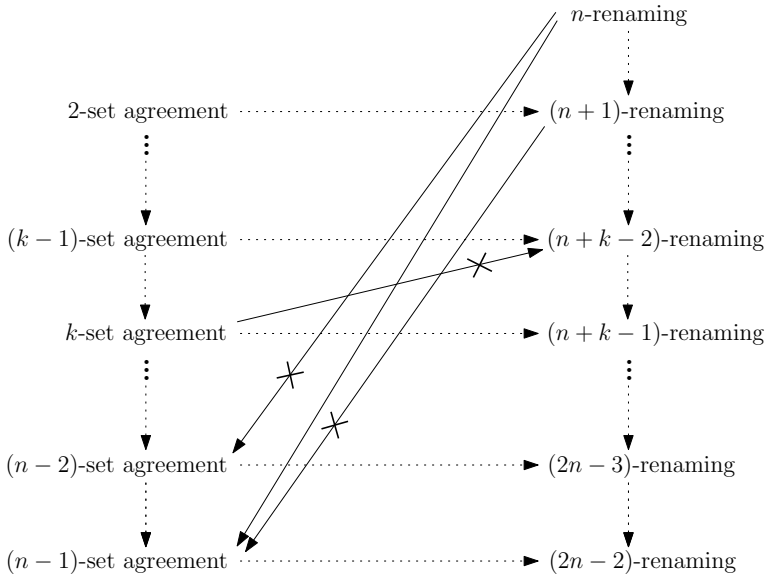

Figure 4: The relation between renaming and set agreement.

```
function choose(v_i):
(01) M[i] ← v_i;                 %Each entry of M is initialized to ⊥
(02) ℓ_i ← 𝒴.choose(i);
(03) if ℓ_i ≠ λ̄
(04)     then decide v_i;
(05)     else s_i ← any j ≠ i such that M[j] ≠ ⊥;
(06)          decide M[s_i]
(07) end if
```

Figure 5: Solving $(n, n-1)$-SA from (n, n)-R.

Consider some $\overline{\lambda} \in \{1, \ldots, n\}$ distinct from λ. We solve $(n, n-1)$-SA using $\mathcal{Y}$ and $\overline{\lambda}$ (see Figure 5), recalling that there is no index-independent requirement for $(n, n-1)$-SA: first each p_i announces its proposal v_i by writing it into $M[i]$ (M is a shared array initialized to $\perp$), then calls $\mathcal{Y}$ with its index i as input and finally decides its proposal if it receives a value distinct from $\overline{\lambda}$; otherwise it decides any $M[j]$, where $j \neq i$ and $M[j] \neq \perp$. In other words, the process that gets $\overline{\lambda}$ from $\mathcal{Y}$ is the only process that does not decide its proposal. Clearly this implementation verifies the validity requirement of $(n, n-1)$-SA. The termination and agreement properties follow from the observation that if a process gets $\overline{\lambda}$ from $\mathcal{Y}$ then there are at least two proposals into M ($\mathcal{Y}$ outputs $\lambda \neq \overline{\lambda}$ in every solo-execution), and thus two processes agree on the same output value. The theorem follows. $\square_{Theorem\ 13}$

Theorem 14. *For every $n \geq 3$, the set agreement $(n, n-1)$-SA task does not SQD-solve the perfect renaming $\langle n, n, 1, 1\rangle$-GSB task.*

Proof. Suppose there is an algorithm $\mathcal{A}$ that solves $\langle n, n, 1, 1\rangle$-GSB from SQD objects that solve $(n, n-1)$-SA. Consider the solo-execution E_s of $\mathcal{A}$ in which p_n participates with identity $N = 2n-1$ (recall that for $\langle n, n, 1, 1\rangle$-GSB, process start with distinct identities in $[1 \ldots N = 2n-1]$). Thus, p_n decides a value $f \in [1 \ldots n]$ in E_s. Let us assume, w.l.o.g., $f = n$. Let S be the set containing all executions of $\mathcal{A}$ that are extensions of E_s, i.e., processes $p_1 \ldots p_{n-1}$ execute computation steps only after p_n decides $f = n$ in E_s. Hence in every $E' \in S$ in which all $p_1 \ldots p_{n-1}$ decide, they decide distinct values in $[1 \ldots n-1]$. Using E_s, we will modify $\mathcal{A}$ in order to obtain an algorithm $\mathcal{B}$ for $p_1 \ldots p_{n-1}$ that read/write wait-free solves perfect renaming $\langle n-1, n-1, 1, 1\rangle$-GSB, which is not possible.

Intuitively, the initial state of $\mathcal{B}$ is the state of $\mathcal{A}$ at the end of E_s and each $(n, n-1)$-SA object in $\mathcal{A}$ is replaced with a read/write wait-free function.

First note that, due to the specification of $(n, n-1)$-SA, each time p_n invokes an $(n, n-1)$-SA object $\mathcal{X}$ in E_s, it receives from $\mathcal{X}$ the value it proposes. Also, the fact that all $(n, n-1)$-SA objects in $\mathcal{A}$ are SQD implies the following for any such object $\mathcal{X}$: (1) if p_n invokes $\mathcal{X}$ in E_s, then when p_i, $1 \leq i \leq n-1$, calls $\mathcal{X}$ in an extension of E_s, it is possible p_i receives the value p_n proposed to $\mathcal{X}$, and (2) if p_n does not call $\mathcal{X}$ in E_s, then when p_i, $1 \leq i \leq n-1$, calls $\mathcal{X}$ in an extension of E_s, it is possible p_i receives the value it proposes (since at most $n-1$ processes call $\mathcal{X}$). Moreover, observe that for every $(n, n-1)$-SA object $\mathcal{X}$ in $\mathcal{A}$, we can compute if p_n invokes $\mathcal{X}$ in E_s.

Algorithm $\mathcal{B}$ is obtained by replacing each $(n, n-1)$-SA object $\mathcal{X}$ in p_i's code, $1 \leq i \leq n-1$ (p_n is suppressed), as follows: if $\mathcal{X}$ is accessed by p_n in E_s, then it is replaced with the constant function that outputs the value proposed (and decided) by p_n, otherwise it is replaced with the identity function that outputs the value p_i proposes; the initial state of the shared memory of $\mathcal{B}$ is the state of the shared memory of $\mathcal{A}$ at the end of E_s. The observations above imply that for any execution E' of $\mathcal{B}$ there is an execution $E'' \in S$ that is the same as E', i.e., in E'', $p_1 \dots p_{n-1}$ decide distinct values in $[1 \dots n-1]$, and hence $\mathcal{B}$ read/write solves $\langle n-1, n-1, 1, 1\rangle$-GSB. However, Corollary 5 shows that $\langle n-1, n-1, 1, 1\rangle$-GSB is not read/write wait-free solvable (see also [7]). A contradiction. $\square_{Theorem\ 14}$

7.2 Perfect renaming cannot solve $(n, n-2)$-SA

This section presents two separation results: when considering SOD objects, perfect renaming cannot solve $(n, n-2)$-SA and renaming $\langle n, n+1, 0, 1\rangle$-GSB cannot solve $(n, n-1)$-SA. In the next subsection we address the case of SQD objects.

Lemma 7. *If there is an* FND *object that solves perfect renaming* $\langle n, n, 1, 1\rangle$-GSB*, then there is an* SOD *object that solves renaming* $\langle n, n+1, 0, 1\rangle$-GSB.

Proof. Let $\mathcal{X}$ be an FND object that solves $\langle n, n, 1, 1\rangle$-GSB. Using $\mathcal{X}$ and an n-dimensional shared array M (each entry is initialized to $\perp$), it can be implemented a SOD object that solves $\langle n, n+1, 0, 1\rangle$-GSB. First, each p_i writes its input in $M[i]$ and then, it reads all M; if p_i sees only its input value in M, then it decides $n+1$, otherwise it calls $\mathcal{X}$ with its input name as input and decides the value it gets from $\mathcal{X}$. Clearly, in every solo-execution the participating process decides $n+1$. $\square_{Lemma\ 7}$

Theorem 15. *For every* n*, the renaming* $\langle n, n+1, 0, 1\rangle$-GSB *task does not SOD-solve the set agreement* $(n, n-1)$-SA *task.*

Proof.
It has been proved that $(n, n-1)$-SA is not read/write wait-free solvable [15, 44, 56], hence $\langle n, n, 1, 1\rangle$-GSB $\nrightarrow_{FND}$ $(n, n-1)$-SA, by Theorem 8. The theorem directly follows from Lemma 7. $\square_{Theorem\ 15}$

Theorems 13 and 15 imply the following separation result. To the best of our knowledge this is the first time this result is formally proved.

Theorem 16.
For every n*, the perfect renaming* $\langle n, n, 1, 1\rangle$-GSB *task is strictly stronger that the renaming* $\langle n, n+1, 0, 1\rangle$-GSB *task under the SOD-solvability relation:* $\langle n, n, 1, 1\rangle$-GSB $\rightarrow_{FND}$ $\langle n, n+1, 0, 1\rangle$-GSB *but* $\langle n, n+1, 0, 1\rangle$-GSB $\nrightarrow_{SOD}$ $\langle n, n, 1, 1\rangle$-GSB.

Lemma 8.
Let T_1 *and* T_2 *be GSB tasks on* n *and* $k+1$ *processes, with* $k+1 < n$*. For* $ZZZ \in \{FND, USD, SOD, SQD\}$*, assume that (1)* $T_2 \nrightarrow_{ZZZ} (k+1, k)$*-SA, and (2) the collection of outputs that* $p_1, \dots, p_{k+1}$ *receive in any invocation to a ZZZ object solving* T_1*, are valid outputs for a ZZZ object solving* T_2*. Then,* $T_1 \nrightarrow_{ZZZ} (n, k)$*-SA.*

Proof. Suppose, for the sake of contradiction, there is an algorithm $\mathcal{A}$ that solves (n,k)-SA from ZZZ objects that solve $T1$. Consider the set of executions S of $\mathcal{A}$ in which only $p_1,\ldots,p_{k+1}$ participate. By assumption for any execution $E \in S$, the collection of outputs that $p_1,\ldots,p_{k+1}$ receive in any invocation to a T_1 object, are valid outputs for T_2. This implies that we can get a new algorithm $\mathcal{B}$ on $k+1$ processes by replacing each T_1 object in p_i's code, $1 \leq i \leq k+1$, with a ZZZ object solving T_2 ($p_{k+2},\ldots,p_n$ are suppressed). Observe that for any execution E of $\mathcal{B}$, there is an execution in S that is the same as E. Moreover, the participating processes decide at most k distinct values in E. Therefore, $\mathcal{B}$ solves $(k+1,k)$-SA from ZZZ objects that solve T_2. This contradicts that $T_2 \nrightarrow_{ZZZ} (k+1,k)$-SA. $\square_{Lemma\ 8}$

Theorem 17. *For every $n \geq 3$, the perfect renaming $\langle n,n,1,1\rangle$-GSB task does not SOD-solve the set agreement $(n,n-2)$-SA task.*

Proof. Note the collection of outputs that $p_1,\ldots,p_{n-1}$ receive in any invocation to an SOD object solving perfect renaming $\langle n,n,1,1\rangle$-GSB, are valid outputs for SOD objects solving renaming $\langle n-1,n,0,1\rangle$-GSB. Also by Theorem 15 on $n-1$ processes, $\langle n-1,n,0,1\rangle$-GSB $\nrightarrow_{SOD}$ $(n-1,n-2)$-SA. From Lemma 8 we get $\langle n,n,1,1\rangle$-GSB $\nrightarrow_{SOD}$ $(n,n-2)$-SA. $\square_{Theorem\ 17}$

7.3 The case of SQD objects

In this section, we present three impossibility results that hold for stronger SQD objects. The first one, Theorem 18, shows that perfect renaming cannot implement most set agreement tasks.

Theorem 18. *For every n,k such that $k \leq \frac{n-1}{2}$, the perfect renaming $\langle n,n,1,1\rangle$-GSB task does not SQD-solve the set agreement (n,k)-SA.*

Proof. Observe that the collection of outputs that $p_1,\ldots,p_{k+1}$ receive in any invocation to an SQD object solving $\langle n,n,1,1\rangle$-GSB, are valid outputs for renaming $\langle k+1,n,0,1\rangle$-GSB. Note that $2(k+1)-1 \leq n$ because $k \leq \frac{n-1}{2}$. It is known that there are read/write wait-free objects solving $\langle k+1,n,0,1\rangle$-GSB when $2(k+1)-1 \leq n$ ([21] presents several such solutions); any of these objects is SQD. Thus, we have that $\langle k+1,n,0,1\rangle$-GSB $\nrightarrow_{SQD}$ $(k+1,k)$-SA, because $(k+1,k)$-SA is not read/write wait-free solvable [15, 44, 56]. The theorem follows from Theorem 8. $\square_{Theorem\ 18}$

Theorems 19 and 20 complement a result in [36] where it is shown that renaming $\langle n,2n-2,0,1\rangle$-GSB cannot implement $(n,n-1)$-SA when n is odd. Theorem 19 shows that $\langle n,2n-2,0,1\rangle$-GSB cannot solve $(n,n-2)$-SA for any value of n, while Theorem 20 extends the result in [36] for every n that is not a power of two.

Theorem 19. *For every n, the renaming $\langle n,2n-2,0,1\rangle$-GSB task does not SQD-solve the set agreement $(n,n-2)$-SA task.*

Proof. The proof is very similar to the proof of Theorem 18. $\square_{Theorem\ 19}$

Theorem 20. *For every n that is not a power of 2, the renaming $\langle n,2n-2,0,1\rangle$-GSB task does not SQD-solve the set agreement $(n,n-1)$-SA task.*

Proof.
It is shown in [36, 35] that if n is odd, $\langle n,2n-2,0,1\rangle$-GSB $\nrightarrow_{SQD}$ $(n,n-1)$-SA. It is proved in [20] that if n is not a power of a prime, then $\langle n,2n-2,0,1\rangle$-GSB is read/write wait-free solvable, hence $\langle n,2n-2,0,1\rangle$-GSB $\nrightarrow_{SQD}$ $(n,n-1)$-SA because $(n,n-1)$-SA is not read/write wait-free solvable [15, 44, 56]. If n is not a power of two, then n is odd or is not power of a prime. The theorem follows. $\square_{Theorem\ 20}$

7.4 From (n, k)-SA to renaming $\langle n, n+k-2, 0, 1\rangle$-GSB

It has been proved that (n, k)-SA can solve $\langle n, n+k-1, 0, 1\rangle$-GSB [31] (the paper implicitly assumes FND objects). Theorem 21 shows that in some cases (n, k)-SA cannot do something better than that, namely, it cannot solve $\langle n, n+k-2, 0, 1\rangle$-GSB.[5]

Theorem 21. *For every n, k where $n > k$ and k is power of a prime number, the set agreement (n, k)-SA task does not SQD-solve the renaming $\langle n, n+k-2, 0, 1\rangle$-GSB task.*

Proof. The proof is a generalization of the proof of Theorem 14. Suppose there is an algorithm $\mathcal{A}$ that solves renaming $\langle n, n+k-2, 0, 1\rangle$-GSB from SQD objects that solve (n, k)-SA. Let E_s be any execution of $\mathcal{A}$ in which only the $n-k$ processes $p_{k+1} \dots p_n$ participate with identities $N-(n-k-1) \dots N$, with $N = 2n-1$. Thus, in E_s, $p_{k+1} \dots p_n$ decide $n-k$ distinct values in $[1 \dots n+k-2]$. Let us assume, w.l.o.g, they decide the values in $[2k-1 \dots n+k-2]$. Let S be the set containing all executions of $\mathcal{A}$ that are extensions of E_s, i.e., processes $p_1 \dots p_k$ execute computation steps only after $p_{k+1} \dots p_n$ decide in E_s. Hence in every $E' \in S$ in which all $p_1 \dots p_k$ decide, they decide distinct values in $[1 \dots 2k-2]$. From $\mathcal{A}$ and E_s, we obtain an algorithm $\mathcal{B}$ for $p_1 \dots p_k$ that read/write wait-free solves renaming $\langle k, 2k-2, 0, 1\rangle$-GSB, which is not possible when k is power of a prime number.

The fact that all (n, k)-SA objects in $\mathcal{A}$ are SQD implies the following for any such object $\mathcal{X}$: (1) if some process invokes $\mathcal{X}$ in E_s, then when p_i, $1 \leq i \leq k$, calls $\mathcal{X}$ in an extension of E_s, it is possible p_i receives any value $\mathcal{X}$ outputs in E_s, and (2) if no process calls $\mathcal{X}$ in E_s, then when p_i, $1 \leq i \leq k$, invokes $\mathcal{X}$ in an extension of E_s, it is possible p_i receives the value it proposes (since at most k processes call $\mathcal{X}$). Moreover, observe that for every (n, k)-SA object $\mathcal{X}$ in $\mathcal{A}$, we can compute if some process $p \in \{p_{k+1} \dots p_n\}$ invokes $\mathcal{X}$ in E_s and the value p receives from $\mathcal{X}$.

Algorithm $\mathcal{B}$ is obtained by replacing each (n, k)-SA object $\mathcal{X}$ in p_i's code, $1 \leq i \leq k$, ($p_{k+1} \dots p_n$ are suppressed) as follows: if $\mathcal{X}$ is accessed by some process in E_s, then it is replaced with a constant function that outputs any value that $\mathcal{X}$ outputs in E_s, otherwise it is replaced with the identity function that outputs the value p_i proposes; the initial state of the shared memory of $\mathcal{B}$ is the state of the shared memory of $\mathcal{A}$ at the end of E_s. The observations above imply that for any execution E' of $\mathcal{B}$ there is an execution $E'' \in S$ that is the same as E', hence $\mathcal{B}$ read/write solves $(k, 2k-2)$-R. However, it is proved in [19] that renaming $\langle k, 2k-2, 0, 1\rangle$-GSB is not read/write wait-free solvable if k is a prime power. A contradiction. $\square_{Theorem\ 21}$

8 GSB vs set agreement

Section 7 proved that perfect renaming is the only member of the renaming family of tasks that is capable to solve $(n-1)$-set agreement. Moreover, perfect renaming is strictly stronger than $(n-1)$-set agreement. In this section we identify a large subfamily of GSB, that we denote $\mathcal{F}$, containing tasks that are strictly stronger than $(n-1)$-set agreement; perfect renaming belongs to $\mathcal{F}$. Since perfect renaming is universal in the GSB family and perfect renaming cannot implement $(n-2)$-set agreement, it follows that no GSB task can implement $(n-2)$-set agreement (at least when considering SOD objects). Thus, the best a task in $\mathcal{F}$ can do is $(n-1)$-set agreement.

This section investigates the internal structure of the tasks in $\mathcal{F}$. We show that for any $n \geq 2$, $\mathcal{F}$ contains infinitely many GSB tasks, and if $n = 2, 3$, all of them are equivalent to perfect renaming. For $n \geq 4$, $\mathcal{F}$ contains a large subfamily $\mathcal{F}'$ of tasks that are strictly weaker than perfect renaming. And a subsubfamily $\mathcal{F}'' \subset \mathcal{F}'$ such that each member of $\mathcal{F}''$ can implement $(n+1)$-renaming. From the fact that $(n+1)$-renaming cannot solve $(n-1)$-set agreement and, as already explained, each task in $\mathcal{F} \supset \mathcal{F}''$ implements $(n-1)$-set agreement, it follows that $(n+1)$-renaming renaming cannot implement any task of $\mathcal{F}''$. Thus, $\mathcal{F}''$ lies exactly between perfect renaming and $(n+1)$-renaming. Figure 6 depicts the relations between the families $\mathcal{F}$, renaming, GSB, and set agreement for $n \geq 4$.

8.1 A noteworthy subfamily $\mathcal{F}$ of GSB

After having defined the task family $\mathcal{F}$, this section shows that, for $n \geq 3$, $(n, n-1)$-SA can be solved from any task in $\mathcal{F}$ while the opposite is not true.

[5] In [31] it is proved that (n, k)-SA cannot solve the stronger adaptive $(p+k-2)$-renaming, for every n and k.

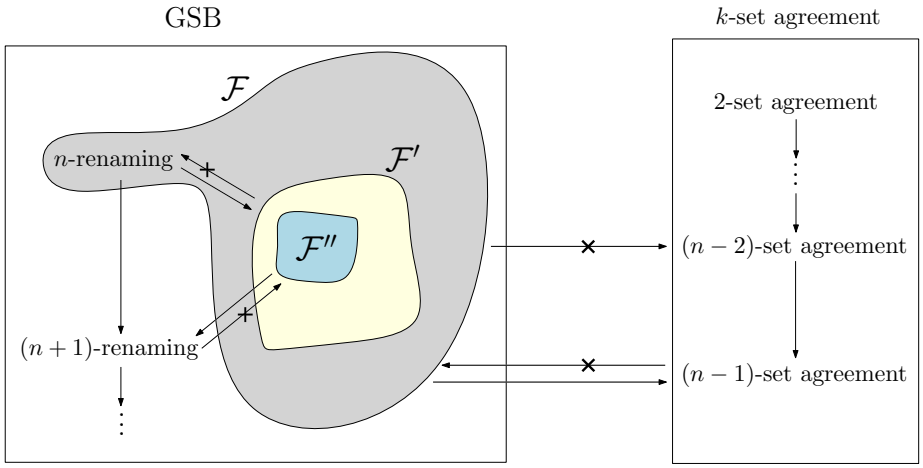

Figure 6: Families $\mathcal{F}$, $\mathcal{F}'$, $\mathcal{F}''$ wrt set agreement, renaming and GSB in systems of $n \geq 4$ processes

Definition 7 (Family $\mathcal{F}$)**.** *A* GSB *task belongs to $\mathcal{F}$ if and only if there are two distinct decision values λ_1 and λ_2 such that in every execution in which all processes decide, exactly one process decides λ_1 and exactly one process decides λ_2.*

Theorem 22. *For every n and $T \in \mathcal{F}$, the GSB task T USD-solves the set agreement $(n, n-1)$-*SA *task.*

Proof. The proof is essentially the same as the proof of Theorem 17. As explained in that proof, we can assume we have a SOD object $\mathcal{Y}$ that solves T. Moreover, there is a value γ such that, in every solo-execution of $\mathcal{Y}$, the participating process p_i obtains γ whatever its input name.

Let λ_1 and λ_2 be values such that in every execution in which all processes invoke and receive a value from $\mathcal{Y}$, exactly one processes gets λ_1 and exactly one process gets λ_2. Consider a value $\overline{\gamma} \in \{\lambda_1, \lambda_2\}$ such that $\overline{\gamma} \neq \gamma$. Using $\mathcal{Y}$ and $\overline{\gamma}$, we implement $(n, n-1)$-SA, recalling that there is no index-independent requirement for $(n, n-1)$-SA. First each p_i announces its proposal v_i by writing it into $M[i]$ (M is a shared array initialized to $\perp$), then calls $\mathcal{Y}$ with its index i as input and finally decides its proposal if it receives a value distinct from $\overline{\gamma}$; otherwise it decides any $M[j]$, where $j \neq i$ and $M[j] \neq \perp$. In other words, the process that gets $\overline{\gamma}$ from $\mathcal{Y}$ is the only process that does not decide its proposal. Clearly this implementation verifies the validity requirement of $(n, n-1)$-SA. The termination and agreement properties follow from the observation that, first, in every execution in which all processes invoke $\mathcal{Y}$, exactly one process gets $\overline{\gamma}$ (since $\overline{\gamma} \in \{\lambda_1, \lambda_2\}$) and, second, if a process gets $\overline{\gamma}$ from $\mathcal{Y}$ then there are at least two proposals into M (because $\mathcal{Y}$ outputs $\gamma \neq \overline{\gamma}$ in every solo-execution). Therefore, if all processes decide, exactly two processes agree on the same output value. $\square_{Theorem\ 22}$

The following task is instrumental in proving that $(n-1)$-set agreement cannot implement any task in $\mathcal{F}$, for $n \geq 3$. The task $\mathcal{U}$ for $n \geq 3$ processes, denoted $\mathcal{U}_n$, is the WSB $\langle n, 2, 1, n-1\rangle$-GSB task with the additional adaptive requirement that in every execution in which at least three processes decide, at least one process decides 1.

Lemma 9. $\forall n \geq 4, \forall T \in \mathcal{F} : \big((n, n-1)\text{-SA} \rightarrow_{SQD} T\big) \Rightarrow \big(\mathcal{U}_{n-1}$ *is read/write wait-free solvable*$\big)$.

Proof. Let $\mathcal{A}$ be an index-independent and wait-free algorithm that solves T from read/write registers and SQD objects that solve $(n, n-1)$-SA. Let $M_{\mathcal{O}}$ be the domain of decision values of T and consider the set S containing every executions of $\mathcal{A}$ in which only processes $p_1, \ldots, p_{n-1}$ participate. Therefore, if a process decides in $E \in S$, it decides a value in $M_{\mathcal{O}}$. From the specification of $(n, n-1)$-SA and since at most only $n-1$ processes participate in every execution in S, it follows that S contains a proper subset S' of executions in which each invocation by p_i, $1 \leq i \leq n-1$, to any $(n, n-1)$-SA object outputs the value p_i uses as input. Note that this does not contradict the assumption that all $(n, n-1)$-SA objects in $\mathcal{A}$ are SQD. This observation allows to obtain an algorithm $\mathcal{B}$ for

$n-1$ processes by modifying p_i's code in $\mathcal{A}$, $1 \leq i \leq n-1$, as follows (p_n is discarded): each invocation to an $(n, n-1)$-SA object is replaced by a function that just outputs the input it receives. Therefore, for any execution E of $\mathcal{B}$, as already explained, there is an execution in S' that is the same as E; hence $\mathcal{B}$ is a read/write wait-free algorithm for $n-1$ processes in which processes decides values in $M_{\mathcal{O}}$. Note that $\mathcal{B}$ is index-independent.

Since $T \in \mathcal{F}$, there exist values λ_1 and λ_2 such that in every execution in which all processes (n processes) decide in T, exactly one processes decides λ_1 and exactly one process decides λ_2. Thus, $\lambda_1, \lambda_2 \in M_{\mathcal{O}}$. Note that if only $n-1$ processes decide in T, then there must be a process that decides either λ_1 or λ_2. In what follows, "all processes" means $p_1, \ldots, p_{n-1}$. The existence of λ_1 and λ_2 imply the following about the values decided by processes in an execution E of $\mathcal{B}$ (which is an execution of $\mathcal{A}$ in which at most $n-1$ processes participate): (1) for each $\lambda \in \{\lambda_1, \lambda_2\}$, at most one process decides λ in E, and (2) if all processes decide in E, then at least one process decides a value in $\{\lambda_1, \lambda_2\}$. From (1) and (2) we conclude the following for every execution E of $\mathcal{B}$:

P1. If at least three processes decide in E (recall $n \geq 4$), then at least one process decides a value in $M_{\mathcal{O}} \setminus \{\lambda_1, \lambda_2\}$.

P2. If all processes decide in E, then at least one process decides a value in $M_{\mathcal{O}} \setminus \{\lambda_1, \lambda_2\}$ and at least one process decides a value in $\{\lambda_1, \lambda_2\}$.

We now modify $\mathcal{B}$ in the following way: each time a process decides a value in $M_{\mathcal{O}} \setminus \{\lambda_1, \lambda_2\}$, its decision is replaced with 1, otherwise its decision is replaced with 2. It is not hard to see that P1 and P2 imply the resulting algorithm $\mathcal{B}$ solves $\mathcal{U}_{n-1}$, and hence $\mathcal{U}_{n-1}$ is read/write wait-free solvable. $\square_{Lemma\ 9}$

Lemma 10. *For every $n \geq 3$, $\mathcal{U}_n$ is not read/write solvable.*

Proof. Suppose, for the sake of contradiction, there exists an index-independent, read/write and wait-free algorithm $\mathcal{A}$ that solves $\mathcal{U}_n$ for $n \geq 3$. Theorem 2 in [45] shows that from $\mathcal{A}$ and read/write registers, it is possible to derive a comparison-based and index-dependent algorithm that solves $\mathcal{U}_n$. The construction is essentially the same to the one in the proof Lemma 6: a comparison-based, read/write, wait-free algorithm that solves $(2n-1)$-renaming, i.e., $\langle n, 2n-1, 0, 1\rangle$-GSB, is used as a preprocessing stage before invoking $\mathcal{A}$; in this way, the resulting algorithm outputs identical values in executions with same interleaving an same relative order on inputs processes. Therefore, we can assume $\mathcal{A}$ is comparison-based.

In [19] it is proved that if n holds a number theoretical property, then WSB is not wait-free solvable from read/write registers (and hence $\langle n, 2n-1, 0, 1\rangle$-GSB is not also, since WSB and $(2n-1)$-renaming, i.e., $\langle n, 2n-1, 0, 1\rangle$-GSB, are FND-equivalent [36]), and it is proved in [20] that if n does not hold that property, then WSB is indeed read/write wait-free solvable. As observed in [11], the number theoretical property has to do with prime powers. Namely, WSB on n processes, i.e., $\langle n, 2, 1, n-1\rangle$-GSB, is read/write solvable if and only if n is not a prime power. Therefore, if n is a prime power, $\mathcal{A}$ cannot exists because clearly $\mathcal{A}$ solves $\langle n, 2, 1, n-1\rangle$-GSB ($\mathcal{U}_n$ is a stronger version of $\langle n, 2, 1, n-1\rangle$-GSB). Thus, in what follows we assume n is not a prime power.

As in the proof of Theorem 12, we use the following technique. Let $\mathcal{B}$ any read/write algorithm in which processes decide either 1 or 2; $\mathcal{B}$ is index-independent and comparison-based. It is known [19] that, for any input configuration I of the system, the number of executions of $\mathcal{B}$ starting from I in which the n processes of the system decide the same value is given by $\#M = 1 + \binom{n}{1}k_1 + , \ldots, + \binom{n}{n-1}k_{n-1}$, for some integers $k_1, \ldots, k_{n-1}$. For $b \in \{1, 2\}$, the value of k_i is completely determined by the number of executions with participating processes $p_1, \ldots, p_i$ (each p_j starts with $I[j]$) in which every participating process decides b (if we consider the other decision value instead of b, the k_i change but the expression still holds).

Thus, for any input configuration, the number of executions of $\mathcal{A}$ in which all processes decide the same value, is given by $\#M = 1 + \binom{n}{1}k_1 + , \ldots, + \binom{n}{n-1}k_{n-1}$, for some integers $k_1, \ldots, k_{n-1}$. By the specification of $\mathcal{U}_n$, it must be that $\#M = 0$. Moreover, $\mathcal{U}_n$ requires that in every execution in which at least three processes decide, at least one process decides 1. Setting $b = 2$, this implies that $k_i = 0$, for $3 \leq i \leq n-1$: in every execution in which only processes $p_1, \ldots, p_i$ participate and decide, at least one of them decides 1, from which follows that there is no execution in which all $p_1, \ldots, p_i$ decide 2. Hence, $k_3, \ldots, k_{n-1} = 0$, and consequently $\#M = 1 + \binom{n}{1}k_1 + \binom{n}{2}k_2 = 0$. It is a standard number theory result that if $\binom{n}{1}$ and $\binom{n}{2}$ have a common factor, then there are no integers k_1, k_2 such that $\#M = 0$. Since n is not a prime power, $n = q_1^{y_1} \cdots q_x^{y_x}$, for some primes $q_1, \ldots, q_x$, $x \geq 2$. Thus,

$\binom{n}{1} = n = q_1^{y_1} \cdots q_x^{y_x}$ and $\binom{n}{2} = \frac{n(n-1)}{2} = \frac{q_1^{y_1} \cdots q_x^{y_x} (q_1^{y_1} \cdots q_x^{y_x} - 1)}{2}$. Note that there is a factor $q_i \neq 2$ of n, and hence $\frac{q_1^{y_1} \cdots q_{i-1}^{y_{i-1}} q_{i+1}^{y_{i+1}} \cdots q_x^{y_x} (q_1^{y_1} \cdots q_x^{y_x} - 1)}{2}$ is an integer. Thus, q_i is factor of both n and $\frac{n(n-1)}{2}$, and consequently, $\#M \neq 0$. A contradiction. $\square_{Lemma\ 10}$

Theorem 23. *For every $n \geq 3$ and $T \in \mathcal{F}$, the set agreement $(n, n-1)$-SA task does not SQD-solve the* GSB *task T.*

Proof. First consider the case $n = 3$. Consider any $T \in \mathcal{F}$ and let λ_1 and λ_2 be values such that if all processes decide, exactly one processes decides λ_1 and exactly one process decides λ_2. We have $T \rightarrow_{FND} \langle n, n, 1, 1\rangle$-GSB: each processes invokes an FND object that solves T, using its input as input to the object, and decides i if it gets λ_i, $i \in \{1, 2\}$, otherwise it decides 3. Also it is proved in [23] that $(n, n-1)$-SA $\nrightarrow_{SOD} \langle n, n, 1, 1\rangle$-GSB, for $n \geq 3$. It is not difficult to show that impossibility proof in [23] also holds for SQD objects, hence $(n, n-1)$-SA $\nrightarrow_{SQD}$ $\langle n, n, 1, 1\rangle$-GSB. Therefore, $(n, n-1)$-SA $\nrightarrow_{SQD} T$, since $T \rightarrow_{FND} \langle n, n, 1, 1\rangle$-GSB.

The case $n \geq 4$ directly follows from Lemmas 9 and 10: if $(n, n-1)$-SA $\rightarrow_{SQD} T$, then $\mathcal{U}_{n-1}$ is read/write wait-free solvable, which is impossible. $\square_{Theorem\ 23}$

8.2 Internal structure of $\mathcal{F}$

It is easy to see that, for every $n \geq 2$, $x \geq 3$, $\langle n, x, [0, \ldots, 0, 1, 1], [1, \ldots, 1, 1, 1]\rangle$-GSB $\in \mathcal{F}$. Therefore, $\mathcal{F}$ contains infinitely many tasks. Moreover, clearly perfect renaming $\langle n, n, 1, 1\rangle$-GSB belong to $\mathcal{F}$, and thus for any $T \in \mathcal{F}$, $\langle n, n, 1, 1\rangle$-GSB $\rightarrow_{FND} T$, because prefect renaming is universal in the GSB, by Theorem 9. For $n = 2, 3$, all tasks in $\mathcal{F}$ are equivalent, as shown in Theorem 24.

Theorem 24. $\forall n = 2, 3, \forall T_1, T_2 \in \mathcal{F} : T_1 \rightarrow_{FND} T_2 \wedge T_2 \rightarrow_{FND} T_1$.

Proof. As already explained, for every $T \in \mathcal{F}$, $\langle n, n, 1, 1\rangle$-GSB $\rightarrow_{FND} T$. Therefore, it is enough to prove the opposite direction, namely, for every $T \in \mathcal{F}, T \rightarrow_{FND} \langle n, n, 1, 1\rangle$-GSB.

Consider any $T \in \mathcal{F}$ and let λ_1, λ_2 be values such that in every execution of T in which all processes decide, exactly one process decides λ_1 and exactly one process decides λ_2. Let $\mathcal{X}$ be an FND object that solves T. Using $\mathcal{X}$, we solve $\langle n, n, 1, 1\rangle$-GSB. For $n = 2$, each process first calls $\mathcal{X}$ using its input as input to $\mathcal{X}$, and then decides i if it obtains λ_i, $i \in \{1, 2\}$, from $\mathcal{X}$. Similarly, for $n = 3$, each process first calls $\mathcal{X}$ using its input as input to $\mathcal{X}$, and then decides i if it obtains λ_i, $i \in \{1, 2\}$, from $\mathcal{X}$, otherwise it decides 3. The correctness of the implementation directly follows from the specification of T. $\square_{Theorem\ 24}$

When $n \geq 4$, $\mathcal{F}$ has a more interesting structure: three subfamilies $\mathcal{F}_1$, $\mathcal{F}_2$ and $\mathcal{F}_3$ of $\mathcal{F}$ are strictly weaker than perfect renaming. Therefore, the computability power of these subfamilies is in between perfect renaming and $(n-1)$-set agreement.

Lemmas 11 and 12 show that some specific tasks of $\mathcal{F}$ are strictly weaker than perfect renaming (to not overload the presentation, their proofs are given in Appendix C). Lemmas 11 considers an asymmetric version of the $(n-k)$-slot task, while Lemma 12 consider an asymmetric version of $(n+1)$-renaming. These two lemmas will prove that $\mathcal{F}_1$, $\mathcal{F}_2$ and $\mathcal{F}_3$ are strictly weaker than perfect renaming. The proofs of these lemmas are interesting in their own since they heavily exploit the non-deterministic properties of SQD objects.

Lemma 11. $\forall n, x : \big(n \geq 4 \wedge 1 \leq x \leq n-3\big) \Rightarrow \big(\langle n, n-x, [1, .., 1, x+1], [1, .., 1, x+1]\rangle$-GSB $\nrightarrow_{SQD}$ $\langle n, n, 1, 1\rangle$-GSB$\big)$.

Lemma 12. $\forall n \geq 4 : \langle n, n+1, [1, 1, 0, .., 0], [1, 1, 1, .., 1]\rangle$-GSB $\nrightarrow_{SQD} \langle n, n, 1, 1\rangle$-GSB.

For every n, x, y, z such that $n \geq 4, 1 \leq x \leq n-4, 1 \leq y \leq x+1$ and $z = x+2-y$, the $\mathcal{F}_1$ contains $\langle n, n-x, [1, 1, 1, .., 1, y], [1, 1, z, .., z, x+1]\rangle$-GSB, denoted $\mathcal{U}_{n,x,y,z}$, which is an asymmetric version of $(n-k)$-slot. Note that the task $\langle n, n-x, [1, .., 1, x+1], [1, .., 1, x+1]\rangle$-GSB in Lemma 11 is $\mathcal{U}_{n,x,x+1,1}$.

For every n, x such that $n \geq 4$ and $x \geq n+1$, $\mathcal{F}_2$ contains the task $\langle n, x, [1,1,0,..,0], [1,1,1,..,1]\rangle$-GSB, denoted $\mathcal{V}_{n,x}$, which is an asymmetric version of x-renaming. The $\langle n, n+1, [1,1,0,..,0], [1,1,1,..,1]\rangle$-GSB task used in Lemma 12 is $\mathcal{V}_{n,n+1}$.

Let $\overrightarrow{\mathbf{1}}^z$ denote the z-dimensional vector $[1,..,1]$, and for $0 \leq x \leq z$, let $\overrightarrow{\mathbf{1}}^z_x$ denote the z-dimensional vector with x 1's at the beginning and the rest with 2's, $[1,..,1,2,..,2]$. For every n, x such that $n \geq 4$ and $2 \leq x \leq n-2$, the subfamily $\mathcal{F}_3$ contains the task $\langle n, n-1, \overrightarrow{\mathbf{1}}^{n-1}, \overrightarrow{\mathbf{1}}^{n-1}_x\rangle$-GSB, which is denoted $\mathcal{W}_{n,x}$.

Theorem 25. $\forall\, \mathcal{U}_{n,x,y,z} \in \mathcal{F}_1 : \mathcal{U}_{n,x,y,z} \nrightarrow_{SQD} \langle n, n, 1, 1\rangle$-GSB.

Proof. Let us observe that $\mathcal{U}_{n,x,y,z}$ is the task $\langle n, n-x, [1,1,1,..,1,y], [1,1,z,..,z,x+1]\rangle$-GSB, where $1 \leq y \leq x+1$ and $z = x+2-y$, and $\mathcal{U}_{n,x,x+1,1}$ is $\langle n, n-x, [1,..,1,x+1], [1,..,1,x+1]\rangle$-GSB. Therefore, $\mathcal{U}_{n,x,x+1,1}$ can FND-solve $\mathcal{U}_{n,x,y,z}$: processes call an FND object that solves $\mathcal{U}_{n,x,x+1,1}$ and decide the values they receive from the object. Hence, $\mathcal{U}_{n,x,x+1,1} \rightarrow_{SQD} \mathcal{U}_{n,x,y,z}$. By Lemma 11, $\mathcal{U}_{n,x,x+1,1} \nrightarrow_{SQD} \langle n, n, 1, 1\rangle$-GSB, from which follows that $\mathcal{U}_{n,x,y,z} \nrightarrow_{SQD} \langle n, n, 1, 1\rangle$-GSB. $\square_{Theorem\ 25}$

Theorem 26. $\forall\, \mathcal{V}_{n,x} \in \mathcal{F}_2 : \mathcal{V}_{n,x} \nrightarrow_{SQD} \langle n, n, 1, 1\rangle$-GSB.

Proof. Clearly, for every $\mathcal{V}_{n,x} \in \mathcal{F}_2$, $\mathcal{V}_{n,n+1} \rightarrow_{FND} \mathcal{V}_{n,x}$, hence $\mathcal{V}_{n,n+1} \rightarrow_{SQD} \mathcal{V}_{n,x}$. Therefore, $\mathcal{V}_{n,x} \nrightarrow_{SQD} \langle n, n, 1, 1\rangle$-GSB, because $\mathcal{V}_{n,n+1} \nrightarrow_{SQD} \langle n, n, 1, 1\rangle$-GSB, by Lemma 12. $\square_{Theorem\ 26}$

Theorem 27. $\forall\, \mathcal{W}_{n,x} \in \mathcal{F}_3 : \mathcal{W}_{n,x} \nrightarrow_{SQD} \langle n, n, 1, 1\rangle$-GSB.

Proof. Consider the task $\mathcal{U}_{n,1,2,1}$, namely, $\langle n, n-1, [1,1,..,1,2], [1,1,..,1,2]\rangle$-GSB. We have that $\mathcal{W}_{n,x}$ is $\langle n, n-1, [1,..,1], [1,..,1,2,..,2]\rangle$-GSB. Thus, clearly $\mathcal{U}_{n,1,2,1} \rightarrow_{FND} \mathcal{W}_{n,x}$, hence $\mathcal{U}_{n,1,2,1} \rightarrow_{SQD} \mathcal{W}_{n,x}$. By Lemma 11, $\mathcal{U}_{n,1,2,1} \nrightarrow_{SQD} \langle n, n, 1, 1\rangle$-GSB, from which follows that $\mathcal{W}_{n,x} \nrightarrow_{SQD} \langle n, n, 1, 1\rangle$-GSB. $\square_{Theorem\ 27}$

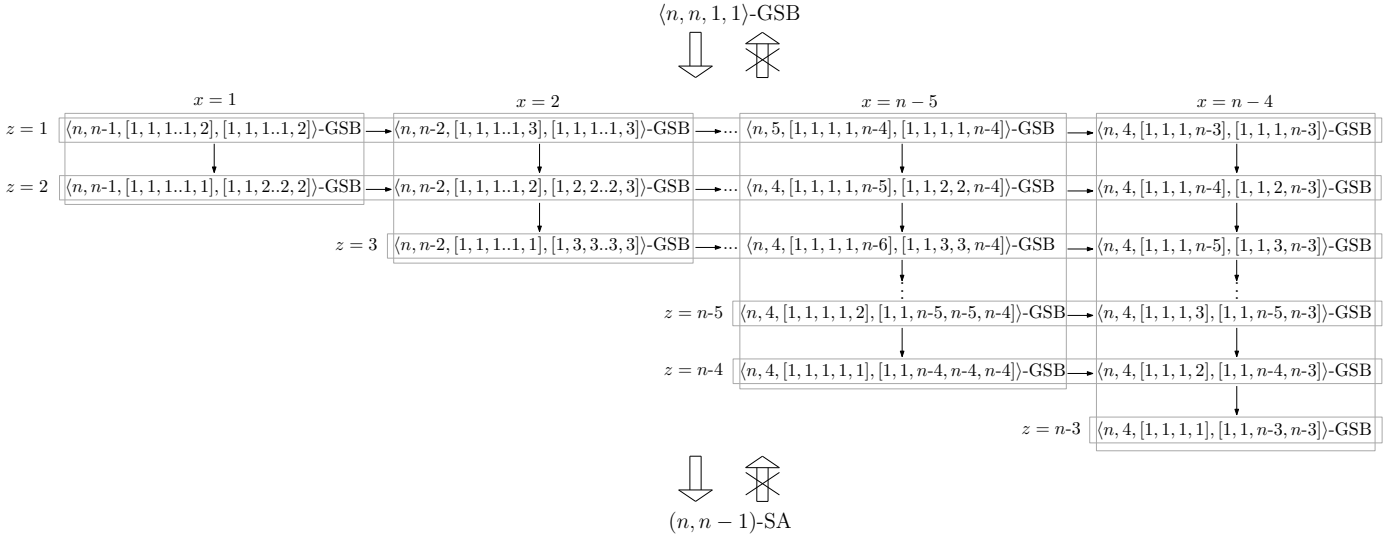

Figure 7: The sub-family $\mathcal{F}_1$

Internal structure of $\mathcal{F}_1$ Figure 7 depicts how the members of $\mathcal{F}_1$ are related under the FND-solvability relation. The vertical arrows are easy to prove. For the horizontal arrows, Figure 7 states $\mathcal{U}_{n,x,y,z} \rightarrow_{FND} \mathcal{U}_{n,x+1,y',z}$. By the definition of $\mathcal{U}_{n,x,y,z}$ and $\mathcal{U}_{n,x+1,y',z}$, we have that $z = x + 2 - y$ and $z = (x + 1) + 2 - y'$, from which follows that $y = y' - 1$. Thus, $\mathcal{U}_{n,x,y,z}$ is the $\langle n, n-x, [1,1,1,..,1,y], [1,1,z,..,z,x+1]\rangle$-GSB task and $\mathcal{U}_{n,x+1,y',z}$ is the $\langle n, n-(x+1), [1,1,1,..,1,y+1], [1,1,z,..,z,(x+1)+1]\rangle$-GSB task. In an implementation of $\mathcal{U}_{n,x+1,y',z}$ based on FND objects, first processes call an FND object $\mathcal{X}$ that solves $\mathcal{U}_{n,x,y,z}$, and then every process decides the value w it receives from $\mathcal{X}$ only if $w \neq n-x$, otherwise it decides $n-(x+1)$. Since $\mathcal{X}$ outputs $n-(x+1)$ at least at one process and $n-x$ at least at y processes, we have that at least $y+1$ processes decide $n-(x+1)$ in the implementation. Also, for each $i \in \{1,\ldots,n-(x+2)\}$, $\mathcal{X}$ outputs i at least at one process, and consequently the number of processes that get either $n-(x+1)$ or $n-x$ from $\mathcal{X}$ is at most $x+2$.

We conjecture that arrows in Figure 7 are strict, namely, for every pair of distinct tasks in $\mathcal{F}_1$, one of them is strictly stronger than the other, even if we consider SQD objects.

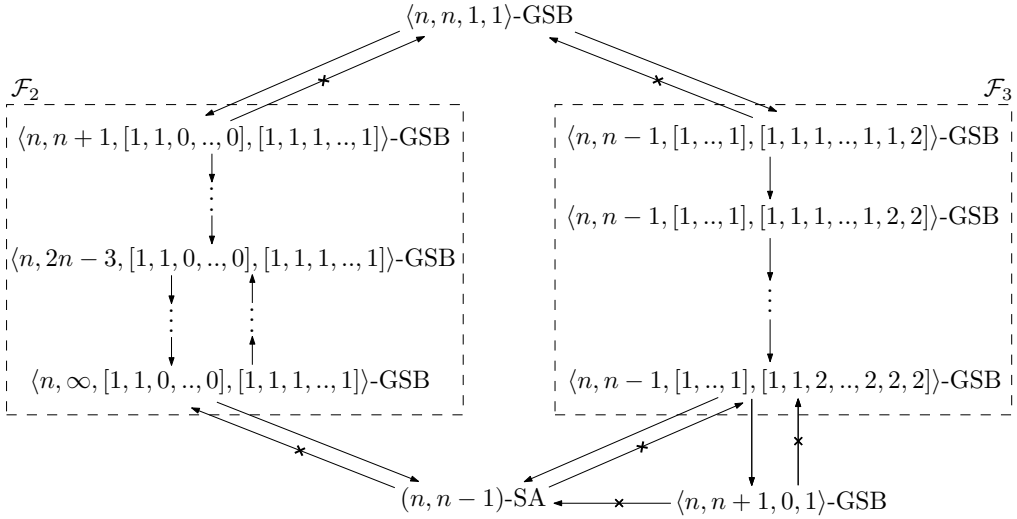

Figure 8: The sub-families $\mathcal{F}_2$ and $\mathcal{F}_3$

Internal structure of $\mathcal{F}_2$ Figure 8 (left) depicts how the members of $\mathcal{F}_2$ are related under the FND-solvability relation. The arrows going down are straightforward to prove. For the arrows going up, Figure 8 states $\mathcal{V}_{n,x+1} \rightarrow_{FND} \mathcal{V}_{n,x}$, where $x \geq 2n-3$. Let $\mathcal{A}$ be any read/write-based wait-free algorithm that solves adaptive $(2p-1)$-renaming ([21] presents many such algorithms), namely, the output space is $[1,..,2p-1]$ in every execution in which $p \leq n$ processes participate. In an implementation of $\mathcal{V}_{n,x}$ based on FND objects, first processes call an FND object $\mathcal{X}$ that solves $\mathcal{V}_{n,x+1}$, and then each process p decides the value u it receives from $\mathcal{X}$ only if $u = 1, 2$, otherwise, using u as input, p invokes $\mathcal{A}$, and decides $w+2$, where w is the value p gets from $\mathcal{A}$. By the definition of $\mathcal{V}_{n,x+1}$, $\mathcal{X}$ outputs 1 at exactly one process and outputs 2 at exactly one process, hence at most $n-2$ processes call $\mathcal{A}$, whose distinct decision values are in $[1+2=3,..,2(n-2)-1+2=2n-3]$ (which is correct since $x \geq 2n-3$). We conjecture that for every $n+1 \leq x \leq 2n-4$, $\mathcal{V}_{n,x+1} \nrightarrow_{SQD} \mathcal{V}_{n,x}$.

Internal structure of $\mathcal{F}_3$ The subfamily$\mathcal{F}_3$ and its relation with $(n, n-1)$-SA and $\langle n, n+1, 0, 1\rangle$-GSB are depicted in Figure 8 (right). The arrows among members of $\mathcal{F}_3$ are under the FND-solvability relation. Clearly each $\mathcal{W}_{n,x} \in \mathcal{F}_3$ FND-solves the $(n-1)$-slot task, namely, $\langle n, n-1, [1,..,1], [2,..,2]\rangle$-GSB. Thus, by Lemma 13 below, $\mathcal{W}_{n,x} \rightarrow_{FND} \langle n, n+1, 0, 1\rangle$-GSB.

Lemma 13. $\forall n, \langle n, n-1, [1,..,1], [2,..,2]\rangle$-GSB $\rightarrow_{FND} \langle n, n+1, 0, 1\rangle$-GSB

Proof. Every process p_i first invokes an FND object $\mathcal{X}$ that solves $\langle n, n-1, [1,..,1], [2,..,2]\rangle$-GSB and then writes $(name_i, x_i)$ into $M[i]$, where $name_i$ is p_i's input name and x_i is what p_i gets from $\mathcal{X}$ (M is shared memory with all its entries initialized to $\perp$). Then, p_i takes an snapshot s_i of M. If there is no $(name_j, x_j)$ in s_i such that $i \neq j$ and $x_i = x_j$, then p_i decides x_i. Otherwise, if $name_i < name_j$, the p_i decides n, otherwise it decides $n+1$.

Clearly, the implementation is wait-free and the processes decide names in $[1, \ldots, n+1]$. Processes decides distinct names because (1) at most two processes get the same name from $\mathcal{X}$ and (2) processes start with distinct input names. $\square_{Lemma\ 13}$

By Theorem 15, $\langle n, n+1, 0, 1\rangle$-GSB $\nrightarrow_{SOD}$ $(n, n-1)$-SA. Also we know that, for every $\mathcal{W}_{n,x} \in \mathcal{F}_3$, $\mathcal{W}_{n,x} \rightarrow_{USD} (n, n-1)$-SA, and hence $\mathcal{W}_{n,x} \rightarrow_{SOD} (n, n-1)$-SA. Thus, $\langle n, n+1, 0, 1\rangle$-GSB $\nrightarrow_{SOD} \mathcal{W}_{n,x}$,

We conjecture that $\langle n, n+1, 0, 1\rangle$-GSB $\nrightarrow_{SQD} \mathcal{W}_{n,x}$ and, for every two distinct members of $\mathcal{F}_3$, one of them is strictly stronger than the other under the SQD-solvability relation.

8.3 From symmetric GSB to set agreement

It can be verified that every member of $\mathcal{F}$ except perfect renaming is asymmetric. Thus the question: does there exist a symmetric GSB task distinct from perfect renaming that can solve $(n-1)$-set agreement? Theorems 28 and 29 show that that a large number of symmetric GSB tasks do not SOD-solve $(n, n-1)$-SA. In fact, the only case that remains to be proved is $\langle n, m, \lfloor \frac{n}{m} \rfloor, \lceil \frac{n}{m} \rceil\rangle$-GSB, where $n \geq 4$ and $2 \leq m \leq n-1$.

Theorem 28. $\forall n, m, \ell, u : (\, m = 1\,) \vee (\, m \geq n+1\,) \vee (\, n = m \wedge \ell = 0 \wedge u \geq 2) \Rightarrow \langle n, m, \ell, u\rangle$-GSB $\nrightarrow_{SOD}$ $(n, n-1)$-SA.

Proof.
If $m = 1$, clearly $\langle n, m, \ell, u\rangle$-GSB is wait-free solvable from read/write registers, and hence $\langle n, m, \ell, u\rangle$-GSB $\nrightarrow_{SOD}$ $(n, n-1)$-SA, because it is known that $(n, n-1)$-SA is not wait-free solvable from read/write registers [15, 44, 56].

Let us now consider the case $m \geq n+1$. If $\langle n, m, \ell, u\rangle$-GSB is not feasible, then we are done. If $\langle n, m, \ell, u\rangle$-GSB is feasible, then it must be that $\ell \times m \leq n \leq u \times m$, by Lemma 2. The fact that $m \geq n+1$ implies that $\ell \times m \geq \ell \times (n+1)$, and hence $\ell = 0$ and $u \geq 1$. Also it is straightforward to see that for every $u \geq 1$ and $m \geq n+1$, we have that $\langle n, n+1, 0, 1\rangle$-GSB $\rightarrow_{FND}$ $\langle n, m, 0, 1\rangle$-GSB and $\langle n, m, 0, 1\rangle$-GSB $\rightarrow_{FND}$ $\langle n, m, 0, u\rangle$-GSB, and hence $\langle n, n+1, 0, 1\rangle$-GSB $\rightarrow_{SOD}$ $\langle n, m, 0, 1\rangle$-GSB and $\langle n, m, 0, 1\rangle$-GSB $\rightarrow_{SOD}$ $\langle n, m, 0, u\rangle$-GSB. Therefore, $\langle n, n+1, 0, 1\rangle$-GSB $\rightarrow_{SOD}$ $\langle n, m, \ell, u\rangle$-GSB. By Theorem 15, we have that $\langle n, n+1, 0, 1\rangle$-GSB $\nrightarrow_{SOD}$ $(n, n-1)$-SA, thus $\langle n, m, \ell, u\rangle$-GSB $\nrightarrow_{SOD}$ $(n, n-1)$-SA.

For the case $n = m$, $\ell = 0$ and $u \geq 2$, observe that $\langle n, n+1, 0, 1\rangle$-GSB $\rightarrow_{FND}$ $\langle n, m, \ell, u\rangle$-GSB: first processes call an $\langle n, n+1, 0, 1\rangle$-GSB object and then a process that receives x from this object decides 1 if $x = n+1$, otherwise decides x. Note that it is correct two processes decide 1 because $u \geq 2$. Thus, $\langle n, n+1, 0, 1\rangle$-GSB $\rightarrow_{SOD}$ $\langle n, m, \ell, u\rangle$-GSB. As already mentioned, we know $\langle n, n+1, 0, 1\rangle$-GSB $\nrightarrow_{SOD}$ $(n, n-1)$-SA, and consequently $\langle n, m, \ell, u\rangle$-GSB $\nrightarrow_{SOD}$ $(n, n-1)$-SA. $\square_{Theorem\ 28}$

Lemma 14. $\forall n, m, \ell, u : 2 \leq m \leq n-1 \wedge \ell = \lfloor \frac{n}{m} \rfloor - 1 \wedge u = \lfloor \frac{n}{m} \rfloor + 1 \Rightarrow \langle n, n+1, 0, 1\rangle$-GSB $\rightarrow_{FND}$ $\langle n, m, \ell, u\rangle$-GSB.

Proof.
In an implementation of an $\langle n, m, \ell, u\rangle$-GSB task, processes first invoke an FND object that solves $\langle n, n+1, 0, 1\rangle$-GSB, and then a process getting y from the object, decides $(y \mod m) + 1$. The correctness proof of this implementation consists in showing that the size of the equivalence classes $[0], \ldots, [m-1]$ induced by mod operator over $1, \ldots, n+1$ are such that, for every $j \in \{0, \ldots, m-1\}$, (1) $\sum_{i=0, i\neq j}^{m-1} \#[i] \leq n - \ell$ and (2) $\#[j] \leq u$, where $\#[j]$ denotes the size of class $[j]$. For any $x \in \{1, , \ldots, , m\}$, (1) implies that there is no execution in which all processes decide, and less than ℓ processes decide x, and (2) implies that there is no execution in which more than u processes decide x. In what follows we prove (1) and (2).

As already mentioned, mod operator splits integers $1, \ldots, n+1$ into m equivalence classes, $[0], \ldots, [m-1]$, each one of them containing roughly the same amount of elements. To be precise, for every $i \in \{0, \ldots, m-1\}$, $\lfloor \frac{n+1}{m} \rfloor \leq \#[i] \leq \lfloor \frac{n+1}{m} \rfloor + 1$. In particular, if $n+1$ is multiple of m, then for all $i \in \{0, \ldots, m-1\}$, $\#[i] = \lfloor \frac{n+1}{m} \rfloor$, otherwise there is at least one $i \in \{0, \ldots, m-1\}$ such that $\#[i] = \lfloor \frac{n+1}{m} \rfloor + 1$.

Clearly we have $\sum_{j=0}^{m-1} \#[j] = n+1$, thus, for each $i \in \{0, \ldots, m-1\}$,

$$\sum_{j=0, j \neq i}^{m-1} \#[j] = \sum_{j=0}^{m-1} \#[j] - \#[i] \leq n + 1 - \lfloor \frac{n+1}{m} \rfloor,$$

because, as already explained, $\#[i] \geq \lfloor \frac{n+1}{m} \rfloor$. Therefore, if we prove $\ell \leq \lfloor \frac{n+1}{m} \rfloor - 1$, then (1) holds. If $n+1$ is a multiple of m, then $\lfloor \frac{n+1}{m} \rfloor = \lfloor \frac{n}{m} \rfloor + 1$, hence $\ell = \lfloor \frac{n}{m} \rfloor - 1 = \lfloor \frac{n+1}{m} \rfloor - 2$. If $n+1$ is not a multiple of m, then $\lfloor \frac{n+1}{m} \rfloor = \lfloor \frac{n}{m} \rfloor$, hence $\ell = \lfloor \frac{n}{m} \rfloor - 1 = \lfloor \frac{n+1}{m} \rfloor - 1$.

To prove (2), first note that if $n+1$ is a multiple of m, then $\lfloor \frac{n+1}{m} \rfloor = \lfloor \frac{n}{m} \rfloor + 1$, and hence, as already mentioned, for each $i \in \{0, \ldots, m-1\}$, $\#[i] = \lfloor \frac{n}{m} \rfloor + 1$. If $n+1$ is not a multiple of m, then $\lfloor \frac{n+1}{m} \rfloor = \lfloor \frac{n}{m} \rfloor$, and thus for every $i \in \{0, \ldots, m-1\}$, $\#[i] \leq \lfloor \frac{n}{m} \rfloor + 1$. $\square_{Lemma\ 14}$

Theorem 29. $\forall n, m, \ell, u : 2 \leq m \leq n-1 \wedge 1 \leq \ell \leq \lfloor \frac{n}{m} \rfloor - 1 \wedge u = n - \ell(m-1) \Rightarrow \langle n, m, \ell, u \rangle$-GSB $\nrightarrow_{SOD}$ $(n, n-1)$-SA.

Proof. Consider a GSB task $\langle n, m, \ell, u \rangle$-GSB as the theorem considers. Let $\ell' = \lfloor \frac{n}{m} \rfloor - 1$ and $u' = \lfloor \frac{n}{m} \rfloor + 1$. By Lemma 14, $\langle n, n+1, 0, 1 \rangle$-GSB $\rightarrow_{FND}$ $\langle n, m, \ell', u' \rangle$-GSB, hence $\langle n, n+1, 0, 1 \rangle$-GSB $\rightarrow_{SOD}$ $\langle n, m, \ell', u' \rangle$-GSB. Thus, $\langle n, m, \ell', u' \rangle$-GSB $\nrightarrow_{SOD}$ $(n, n-1)$-SA, since $\langle n, n+1, 0, 1 \rangle$-GSB $\nrightarrow_{SOD}$ $(n, n-1)$-SA, by Theorem 15. It is straightforward to prove that $\langle n, m, \ell', u' \rangle$-GSB $\rightarrow_{SOD}$ $\langle n, m, \ell, u \rangle$-GSB, and thus $\langle n, m, \ell, u \rangle$-GSB $\nrightarrow_{SOD}$ $(n, n-1)$-SA. $\square_{Theorem\ 29}$

9 Conclusions and open questions

This paper studied the problem of breaking symmetry in the standard asynchronous wait-free read/write crash-prone model of computation. One of its main contributions is the definition of the conceptual framework of GSB tasks. The GSB family contains tasks like renaming and WSB, which are considered fundamental in the theory of distributed computing, as well as other tasks that are interesting on their own, like the k-WSB task (a stronger version of WSB), and the k-slot task. A major result is that perfect renaming (i.e. n-renaming) is universal in GSB, namely, it can solve any other GSB task.

After studying the basic properties of GSB tasks, we focused on comparing the computability power of GSB with set agreement, a family of tasks that model the problem of reaching agreement in a distributed system. In order to do that, we first introduced four variants of non-determinism of a concurrent shared object. These forms of non-determinism induce a solvability hierarchy. Some of these properties have been implicitly used in the past, however here we formally defined them and noticed that they are needed for making a "fair" comparison of GSB and set agreement. Moreover, some of them naturally appear in the absence of randomness in a distributed system.

A main contribution is that perfect renaming, the most powerful GSB task, is strictly stronger than $(n-1)$-set agreement, the weakest set agreement task. Moreover, this is the best perfect renaming can do, since it cannot solve $(n-2)$-set agreement, at least when considering SOD objects. Therefore, in the best case, a GSB task can solve $(n-1)$-set agreement, but not more than that. This result shows that breaking symmetry does not provide much power to reach agreement. In the opposite direction it is known that k-set agreement allows to solve $(n+k-1)$-renaming, as well as other GSB tasks. We also presented several results that complement previous research relating set agreement and renaming.

Then, we showed that perfect renaming is a member of a large family $\mathcal{F}$ of GSB tasks such that each of them is strictly stronger than $(n-1)$-set agreement. Since perfect renaming is universal in GSB and it cannot solve $(n-2)$-set

agreement, no member of $\mathcal{F}$ can solve $(n-2)$-set agreement either. The family $\mathcal{F}$ has an interesting internal structure: it contains a subfamily whose computability power lies in between perfect renaming and $(n+1)$-renaming. This shows that GSB is a "dense" family of natural tasks whose computability power cannot be captured by the renaming family, which has been used intensively in the past as a paradigm for studying the problem of breaking symmetry.

Interestingly, all members of $\mathcal{F}$ but perfect renaming are asymmetric GSB tasks, namely, distinct output values have different constraints about the number of processes that can decide that value. This result seems to suggest that in order to reach some level of agreement, a GSB task has to possess some asymmetry on the output values. Observe that in symmetric GSB tasks there are two sources of symmetry: one from the processes, which comes from the index independent requirement on algorithms, and a second source that comes from the symmetry on the output values.

We believe that the GSB family may lead to a better understanding of the notion of *breaking symmetry* in a distributed system. We see this research as a starting point of a more systematic study of this problem.

Open problems The are many open problems. The following is just a partial list.

- Is there a GSB task that can solve $(n-1)$-set agreement from USD objects and does not belong to $\mathcal{F}$? We believe $\mathcal{F}$ contains all GSB tasks that are capable to solve $(n-1)$-set agreement, even if we consider stronger SQD objects.
- Does $\mathcal{F}$ contain tasks that are incomparable? We conjecture that if $n \geq 5$, $\mathcal{F}$ contains a non-small sub-family of tasks such that every two distinct member are incomparable.
- Consider a task $T \in \mathcal{F}$ and let $\mathcal{S}_{ZZZ}$ be the set containing all tasks that T can ZZZ-solve, ZZZ $\in$ {FND, USD, SOD, SQD}. We have that $\mathcal{S}_{FND} \subseteq \mathcal{S}_{USD} \subseteq \mathcal{S}_{SOD} \subseteq \mathcal{S}_{SQD}$. Lemma 6 shows that indeed $\mathcal{S}_{SOD} \subseteq \mathcal{S}_{USD}$, hence $\mathcal{S}_{USD} = \mathcal{S}_{SOD}$. By Theorem 8, no GSB task can FND-solve $(n-1)$-set agreement, hence $\mathcal{S}_{FND} \neq \mathcal{S}_{USD}$, since T can indeed solve $(n-1)$-set agreement. Is $\mathcal{S}_{USD} \neq \mathcal{S}_{SQD}$? That is, is there a task that T can SQD-solve but cannot USD-solve?
- Is there a symmetric GSB task other than perfect renaming that can USD-solve $(n-1)$-set agreement? We conjecture that the only symmetric GSB task that can solve $(n-1)$-set agreement is perfect renaming, even when considering SQD objects.
- Some of our impossibility results hold for SOD objects. These results depend on Theorem 15 stating that $(n+1)$-renaming does not SOD-solve $(n-1)$-set agreement. Extending Theorem 15 to SQD objects would extend all our impossibility results to SQD objects.

  Using a similar idea as in the proof of Theorem 8, one can show that $(n+1)$-renaming cannot SQD-solve $(n-1)$-set agreement, when considering a restricted class of algorithms in which each process uses the same input in every invocation of an $(n+1)$-renaming object; the input of each process can be its index or an intermediate name obtained from a preprocessing stage. In such a proof, it is enough to consider all executions starting from a single input configuration in which each $(n+1)$-renaming object is replaced with a function that, for each process, returns a fixed output name in every invocation (here is where the proof uses the assumption about the algorithms). It is known that it is impossible to achieve $(n-1)$-set agreement from only read/write register starting from a single input configuration, thus reaching a contradiction.

  This result implies that all our impossibility result that rely on Theorem 15 (for example, Theorems 17, 28 and 29, or that no member of $\mathcal{F}$ can SOD-solve $(n+1)$-renaming), hold for SQD objects and the restricted class of algorithms. However, the general case is still open.

References

[1] Afek Y., Attiya H., Dolev D, Gafni E., Merritt M and Shavit N., Atomic Snapshots of Shared Memory. *Journal of the ACM* 40(4): 873-890 (1993).

[2] Afek Y., Gafni E. and Lieber., Tight Group Renaming on Groups of Size g Is Equivalent to g-Consensus. *Proc. 23rd Int'l Symposium on Distributed Computing (DISC'09)*, Springer LNCS #5805, pp. 111-126, 2009.

[3] Afek Y., Gamzu I., Levy I., Merritt M., and Taubenfeld G., Group Renaming. *Proc. 12th International Conference on Principles of Distributed Systems (OPODIS'08)*, Springer LNCS #5401, pp. 58-72, 2008.

[4] Afek Y., Gafni E., Rajsbaum S., Raynal M. and Travers C., The k-Simultaneous Consensus Problem. *Distributed Computing*, 22:185-195, 2010.

[5] Afek Y. and Merritt M., Fast, Wait-Free $(2k-1)$-Renaming. *Proc. 18th ACM Symposium on Principles of Distributed Computing (PODC'99)*, ACM Press, pp. 105-112, 1999.

[6] Angluin D., Local and Global Properties in Networks of Processors. *Proc. 12th ACM Symposium on Theory of Computing (STOC'80)*, ACM Press, pp. 82-93, 1980.

[7] Attiya H., Bar-Noy A., Dolev D., Peleg D. and Reischuck R., Renaming in an Asynchronous Environment. *Journal of the ACM*, 37(3):524-548, 1990.

[8] Attiya H. and Fouren A., Polynomial and Adaptive Long-lived $(2p-1)$-Renaming. *Proc. 14th Int'l Symposium on Distributed Computing (DISC'00)*, Springer LNCS #1914 , pp.149-163, 2000.

[9] Attiya H., Gorbach A. and Moran S., Computing in Totally Anonymous Asynchronous Shared Memory Systems. *Information and Computation*, 173(2):162–183, 2002.

[10] Attiya H. and Rajsbaum S., The Combinatorial Structure of Wait-Free Solvable Tasks. *SIAM Journal on Computing*, 31(4):1286-1313, 2002.

[11] H. Attiya and A. Paz. Counting-Based Impossibility Proofs for Renaming and Set Agreement. *Proc. 26th Int'l Symposium on Distributed Computing (DISC'12)*, Springer LNCS #7611, pp.356-370, 2012

[12] Attiya H. and Welch J., *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, (2d Edition), Wiley-Interscience, 414 pages, 2004.

[13] Armoni M. and Ben-Ari M., The concept of nondeterminism: its development and implications for teaching. *ACM SIGCSE Bulletin*. 41(2): 141-160, 2009.

[14] Burns, J., Symmetry in Systems of Asynchronous Processes. *22nd IEEE Symposium on Foundations of Computer Science (FOCS'81)*, IEEE Computer Press, 169-174, 1981.

[15] Borowsky E. and Gafni E., Generalized FLP Impossibility Result for t-Resilient Asynchronous Computations. *Proc. 25th ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 91-100, 1993.

[16] Borowsky E. and Gafni E., A Simple Algorithmically Reasoned Characterization of Wait-Free Computations (Extended Abstract). *Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, pp. 189-198, 1997.

[17] Castañeda A., A Study of the Wait-free Solvability of Weak Symmetry Breaking and Renaming. PhD Thesis, Posgrado en Ciencia e Ingeniería de la Computación, UNAM, Mexico, December 2010.

[18] Castañeda A. and Rajsbaum S., New Combinatorial Topology Upper and Lower Bounds for Renaming. *Proc. 27th ACM Symposium on Principles of Distributed Computing (PODC'08)*, ACM Press, pp. 295-304, 2008.

[19] Castañeda A. and Rajsbaum S. New Combinatorial Topology Upper and Lower Bounds for Renaming: The Lower Bound. *Distributed Computing* 22(5-6):287-301, 2010.

[20] Castañeda A. and Rajsbaum S. New Combinatorial Topology Upper and Lower Bounds for Renaming: The Upper Bound. *Journal of the ACM* 59(1): 3, 2012.

[21] Castañeda A., Rajsbaum S. and Raynal M., The Renaming Problem in Shared Memory Systems: an Introduction. *Computer Science Review*, 5(3): 229-251, 2011.

[22] Castañeda A., Rajsbaum S. and Raynal M., Agreement via Symmetry Breaking: On the Structure of Weak Subconsensus Tasks. *Proc. 27th IEEE Int'l Symposium on Parallel and Distributed Processing (IPDPS'13)*, IEEE Computer Press, 1147-1158, 2013.

[23] Castañeda A., Imbs D., Rajsbaum S. and Raynal M., Renaming is Weaker than Set Agreement but for Perfect Renaming: A Map of Sub-Consensus Tasks. *Proc. 10th Latin American Theoretical INformatics Symposium (LATIN 2012)*, Springer LNCS #7256, pp. 145-156, 2012.

[24] Chaudhuri S., More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Inf. and Computation*, 105(1):132-158, 1993.

[25] Chaudhuri S., and Reiners, P., Understanding the Set Consensus Partial Order Using the Borowsky-Gafni Simulation (Extended Abstract). *10th Int'l Workshop on Distributed Algorithms (WDAG'96)*, Springer LNCS #1171, pp. 362-379, 1996.

[26] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.

[27] Dijkstra, E.W., Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):569, 1965.

[28] Dolev D., Lynch N., Pinter S., Stark E. and Weihl W. Reaching Approximate Agreement in the Presence of Faults. *Journal of the ACM*, 33(3):499–516, 1986.

[29] Dinitz Y., Moran S. and Rajsbaum S., Bit complexity of Breaking and Achieving Symmetry in Chains and Rings. *Journal of the ACM*, 55(1), article 3, 32 pages, 2008.

[30] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.

[31] Gafni E., Renaming with k-set Consensus: an Optimal Algorithm in $n+k-1$ Slots. *Proc. 10th Int'l Conference On Principles Of Distributed Systems (OPODIS'06)*, Springer LNCS #4305, pp. 36-44, 2006.

[32] Gafni E., The 01-Exclusion Families of Tasks. *Proc. 12th Int'l Conference on Principles of Distributed Systems (OPODIS'08)*, Springer LNCS #5401, pp. 246-258, 2008.

[33] Gafni E. and Kuznetsov P., On Set Consensus Numbers. *Distributed Computing*, 24:149-163, 2011.

[34] Gafni E. and Rajsbaum S., Musical Benches. *19th International Symposium on Distributed Computing (DISC'05)*, Springer LNCS #3724, pp. 63-77, 2005.

[35] Gafni E. and Rajsbaum S., Distributed Programming with Tasks. *Proc. 10th Int'l Conference on Principles of Distributed Systems (OPODIS'10)*, Springer LNCS #6490, pp. 205-218, 2010.

[36] Gafni E., Rajsbaum S. and Herlihy M., Subconsensus Tasks: Renaming is Weaker Than Set Agreement. *Proc. 20th Int'l Symposium on Distributed Computing (DISC'06)*, Springer LNCS #4167, pp.329-338, 2006.

[37] Gafni E., Rajsbaum S., Raynal M. and Travers C., The Committee Decision Problem. *Proc. Latin American Theoretical Informatics Symposium (LATIN'06)*. Springer LNCS #3887, pp. 502-514, 2006.

[38] Gafni E., Raynal M. and Travers C., Test&Set, Adaptive Renaming and Set Agreement: a Guided Visit to Asynchronous Computability. *Proc. 26th Int'l IEEE Symposium on Reliable Distributed Systems (SRDS 2007)*, IEEE Press, pp. 93-103, 2007.

[39] Gafni E., Mostéfaoui M., Raynal M., and Travers C., From Adaptive Renaming to Set Agreement. *Theoretical Computer Science*, 410(14-15):1328-1335, 2009.

[40] Herlihy M., Wait-Free Synchronization. *ACM Transactions Programming Languages and Systems*, 13(1):124-149, 1991.

[41] Herlihy M.P. and Rajsbaum S., Set Consensus Using Arbitrary Objects (Preliminary Version). *Proc. 13th ACM Symposium on Principles on Distributed Computing (PODC'94)*, ACM Press, pp. 324-333, 1994.

[42] Herlihy M. and Rajsbaum S. The Decidability of Distributed Decision Tasks. *Proc. 29th ACM Symposium on Theory of Computing (STOC'97)*, ACM Press, pp. 589-598, 1997.

[43] Herlihy M.P. and Rajsbaum S., A Classification of Wait-free Loop Agreement Tasks. *Theoretical Computer Science*, 291(1):55-77, 2003.

[44] Herlihy M.P. and Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923, 1999.

[45] Imbs D., Rajsbaum S. and Raynal M., The Universe of Symmetry Breaking Tasks. *Proc. 18th International Colloquium on Structural Information and Communication Complexity (SIROCCO'11)*, Springer LNCS #6796, pp. 66-77, 2011.

[46] Jayanti, P., and Toueg, S. Wakeup under Read/Write Atomicity. *Proc. 4th Int'l Workshop on Distributed Algorithms (WDAG'90)*, Springer LNCS #486, pp. 277–288, 1990.

[47] Lamport L., A Fast Mutual Exclusion Algorithm. *ACM Trans. on Computer Systems*, 5(1):1-11, 1987.

[48] Lamport. L., The Part-time Parliament. *ACM Transactions on Computer Systems*, 16(2):133-169, 1998.

[49] Lynch N.A., *Distributed Algorithms*. Morgan Kaufmann Pub., San Francisco (CA), 872 pages, 1996.

[50] Moir M. and Anderson J., Wait-Free Algorithms for Fast, Long-Lived Renaming. *Science of Computer Programming*, 25(1):1-39, 1995.

[51] Mostéfaoui M., Raynal M., and Travers C., Exploring Gafni's Reduction Land: from Ω^k to Wait-free adaptive $(2p - \lceil \frac{p}{k} \rceil)$-renaming via k-set Agreement. *Proc. 20th Int'l Symposium on Distributed Computing (DISC'09)*, Springer LNCS #4167, pp. 1-15, 2006.

[52] Moran, S., and Wolfsthal, Y., An extended Impossibility Result for Asynchronous Complete Networks. *Information Processing Letters* 26:141-151, 1987.

[53] Raynal M., Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems. *Morgan & Claypool Publishers*, 251 pages, 2010 (ISBN 978-1-60845-293-4).

[54] Raynal M., *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer, 500 pages, 2012 (ISBN 978-3-642-32026-2).

[55] Styer E., and Peterson G. L., Tight Bounds for Shared Memory Symmetric Mutual Exclusion Problems. *Proc. 8th ACM Symposium on Principles of Distributed Computing (PODC'89)*, ACM Press, pp. 177-192, 1989.

[56] Saks M. and Zaharoglou F., Wait-Free k-Set Agreement Is Impossible: The Topology of Public Knowledge. *SIAM J. on Computing*, 29(5):1449-1483, 2000.

A Modeling Tasks and Algorithms Using Topology

Topology definitions: vertices, simplexes and complexes A *simplex* σ is a finite set. The elements of a simplex are its *vertexes*. The dimension of a simplex σ is the number of its vertexes minus 1. If σ has $n+1$ vertexes then it is called an n*-simplex*. A simplex τ is a *face* of σ if τ is a subset of σ. If τ is not equal to σ then τ is a *proper face* of σ. A *complex* $\mathcal{K}$ is a set of simplexes, closed under containment. The dimension of a complex $\mathcal{K}$ is the maximum dimension

of its simplexes. A complex $\mathcal{K}$ of dimension n is an *n-complex*. In what follows we only consider n-complexes in which for every simplex τ, there is an n-simplex τ' that contains τ. For a simplex σ, we often denote as σ the complex containing all faces of σ (including σ itself). A complex $\mathcal{L}$ is a *subcomplex* of the complex $\mathcal{K}$ if $\mathcal{L} \subseteq \mathcal{K}$.

For a domain of inputs I, the *input complex* $\mathcal{I}$ is an $(n-1)$-complex that contains $(n-1)$-simplexes (subsets with n elements) of $\{1, , \ldots, , n\} \times I$, and all their faces, such that no pair of vertexes have the same index, the first entry of each pair. An *output complex*, $\mathcal{O}$, over a domain O, is defined similarly. The meaning of a vertex (i, v) of $\mathcal{I}$ (resp. $\mathcal{O}$) is that process with index i has input (resp. output) v.

Topological definition of a task A *task* is a triple $\langle \mathcal{I}, \mathcal{O}, \triangle \rangle$, where $\mathcal{I}$ is an input complex, $\mathcal{O}$ is an output complex and $\triangle$ is a recursive map carrying each m-simplex σ of $\mathcal{I}$, $0 \leq m \leq n-1$, to a non-empty m-subcomplex of $\mathcal{O}$. This definition has the following interpretation: $\triangle(\sigma)$ is the set of legal final states in executions where only the $m+1$ processes in σ participate.

Topological definition of solving a task Every algorithm has an associated *algorithm complex* $\mathcal{A}$, in which each vertex is labeled with a process id and that process's final state (its *view*). Each simplex thus corresponds to an equivalence class of executions that "look the same" to the processes at its vertexes. The algorithm complex corresponding to executions starting from an input simplex σ is denoted $\mathcal{A}(\sigma)$.

A *vertex map* carries vertexes of one complex to vertexes of another. A *simplicial map* is a vertex map that preserves simplexes. Let $\mathcal{A}$ be the algorithm complex of an algorithm. The algorithm *solves* a task $\langle \mathcal{I}, \mathcal{O}, \triangle \rangle$ if and only if there exists an *id-preserving* simplicial map (i.e., maps vertexes with same id) $\delta : \mathcal{A} \to \mathcal{O}$, called a *decision map*, such that for every simplex $\sigma \in \mathcal{I}$, $\delta(\mathcal{A}(\sigma)) \subset \triangle(\sigma)$.

B Manifold tasks

Manifold tasks are defined using concepts of combinatorial topology (see Appendix A).

Let $\mathcal{K}$ be a complex. $\mathcal{K}$ is a *manifold* if (1) every simplex belongs to at least one n-simplex, and (2) every $(n-1)$-simplex belong to exactly one or two n-simplexes. The *boundary* of $\mathcal{K}$, denoted $\partial\mathcal{K}$, is the subcomplex containing all $(n-1)$-simplexes, an all its faces, that belongs to exactly one $n-1$-simplex.

In a *manifold task* $T = \langle \mathcal{I}, \mathcal{O}, \Delta \rangle$, for each input m-simplex $\sigma \in \mathcal{I}$, $\Delta(\sigma)$ is an m-manifold with $\partial\Delta(\sigma) = \Delta(\partial\sigma)$. Therefore, for each $(m-1)$-face σ' of σ, $\Delta(\sigma') \subset \partial\Delta(\sigma)$, hence for every $(m-1)$-simplex $\tau \in \Delta(\sigma')$, we have that $\tau \in \partial\Delta(\sigma)$, and consequently, there is exactly one m-simplex in $\Delta(\sigma)$ that contains τ, because $\Delta(\sigma)$ is a manifold. From an operational point of view, this has the following interpretation.

Let p be the unique process in $\sigma \setminus \sigma'$. Consider an execution in which first the processes in σ' call (concurrently or not) an object $\mathcal{X}$ that solves T. Hence $\mathcal{X}$ produces outputs at processes in σ' (if the invocations are concurrent, then $\mathcal{X}$ behaves non-deterministically) and accordingly changes its internal state. Then p calls alone $\mathcal{X}$ (after all invocations of processes in σ' have finished). The very definition of T implies that there is only one value $\mathcal{X}$ can output at p, according with its internal state and p's input. As explained above, the values $\mathcal{X}$ produced at processes in σ' correspond to an $(m-1)$-simplex τ in $\partial\Delta(\sigma)$, and hence there is only one m-simplex in $\Delta(\sigma)$ that contains τ, which means that there is one value p can receive from $\mathcal{X}$ that is compatible with the values processes in σ' received from $\mathcal{X}$. Therefore, the very definition of manifold tasks imply that $\mathcal{X}$ is sequentially deterministic (SQD).

C Proof of Lemmas 11 and 12

Lemma 11 $\forall n, x : \big(n \geq 4 \wedge 1 \leq x \leq n-3\big) \Rightarrow \big(\langle n, n-x, [1,..,1,x+1], [1,..,1,x+1]\rangle\text{-GSB} \nrightarrow_{SQD} \langle n, n, 1, 1\rangle\text{-GSB}\big)$.

Proof. Suppose there is a wait-free algorithm $\mathcal{A}$ that solves $\langle n, n, 1, 1\rangle$-GSB from SQD objects that solve $\langle n, n-x, [1,..,1,x+1], [1,..,1,x+1]\rangle$-GSB. Consider the set S containing all executions of $\mathcal{A}$ in which only the $n-(x+1)$ processes $p_{x+2}, \ldots, p_n$ participate, with identities $N-(x+1)-1, \ldots, N$ (recall that for GSB tasks,

processes start with distinct identities in $[1, .., N]$, where $N \geq 2n - 1$). Therefore, in every execution of S in which all $p_{x+2}, \ldots, p_n$ decide, they decide $n - (x + 1)$ distinct values in the range $[1, .., n]$.

Now note that the fact that $\mathcal{A}$ is wait-free and every $\langle n, n - x, [1, .., 1, x + 1], [1, .., 1, x + 1]\rangle$-GSB object in $\mathcal{A}$ is SQD, imply there must exist an $E \in S$ such that (1) all processes $p_{x+2}, \ldots, p_n$ decide, (2) $p_{x+2}, \ldots, p_n$ execute $\mathcal{A}$ sequentially in some order $p_{x_1}, \ldots, p_{x_{n-(x+1)}}$, i.e., $p_{x_{i+1}}$ executes computation steps only after p_{x_i} decides, (3) every $\langle n, n - x, [1, .., 1, x + 1], [1, .., 1, x + 1]\rangle$-GSB object $\mathcal{X}$ outputs i in its i-th (sequential) invocation, whatever the input of the invoking process; hence $\mathcal{X}$ only outputs values in the range $[1, \ldots, n - (x + 1) = n - x - 1]$.

As already mentioned, in E, processes decide $n - (x + 1)$ distinct values in the range $[1, .., n]$. Let $[z_1, .., z_{x+1}]$ be the values that no process decides in E. Let S' be the set containing all executions of $\mathcal{A}$ that are extensions of E, i.e., $p_1, \ldots, p_{x+1}$ execute computation steps only after $p_{x+2}, \ldots, p_n$ decide in E. Hence in every $E' \in S'$ in which all $p_1, \ldots, p_{x+1}$ decide, they decide distinct values in $[z_1, .., z_{x+1}]$. Essentially, $p_1, \ldots, p_{x+1}$ solve $\langle x + 1, x + 1, 1, 1\rangle$-GSB on the space $[z_1, .., z_{x+1}]$, with help of SQD objects that solve $\langle n, n - x, [1, .., 1, x + 1], [1, .., 1, x + 1]\rangle$-GSB. Using E, we modify $\mathcal{A}$ in order to obtain a read/write-based algorithm $\mathcal{B}$ for $p_1, \ldots, p_{x+1}$ that wait-free solves $\langle x + 1, x + 1, 1, 1\rangle$-GSB, which is not possible.

Processes in $\mathcal{B}$, $p_1, \ldots, p_{x+1}$, follow the same state machine as in $\mathcal{A}$, respectively, and the initial state of the shared memory of $\mathcal{B}$ is the state of the shared memory of $\mathcal{A}$ at the end of E. Also, each $\langle n, n - x, [1, .., 1, x + 1], [1, .., 1, x + 1]\rangle$-GSB object $\mathcal{X}$ in p_i's code, $1 \leq i \leq x + 1$, is replaced with a read/write-based wait-free function $\mathsf{f}(\cdot)$. The tricky part is that we have to pick a function $\mathsf{f}(\cdot)$ that follows the behavior of $\langle n, n - x, [1, .., 1, x + 1], [1, .., 1, x + 1]\rangle$-GSB objects in E, namely, each SQD object $\mathcal{X}$ outputs i in i-th sequential invocation, $1 \leq i \leq n - (x + 1)$ (note that $\mathcal{X}$ is not necessarily invoked $n - (x + 1)$ times at the end of E). In this way $p_1, \ldots, p_{x+1}$ cannot distinguish they are dealing with $\mathsf{f}(\cdot)$ and not with a genuine SQD objects that solve $\langle n, n - x, [1, .., 1, x + 1], [1, .., 1, x + 1]\rangle$-GSB. In the end, each execution of $\mathcal{B}$ corresponds to an execution in S'.

Two more observations. (1) From the specification of $\langle n, n - x, [1, .., 1, x + 1], [1, .., 1, x + 1]\rangle$-GSB and since SQD objects can behave non-deterministically in presence of concurrency, it follows that for any $\langle n, n-x, [1, .., 1, x+1], [1, .., 1, x + 1]\rangle$-GSB object $\mathcal{X}$, in every extension E' of E, $\mathcal{X}$ can output $n - x$ at all processes in concurrent invocations (because at most $x + 1$ processes invoke $\mathcal{X}$ in E' and $\mathcal{X}$ can output $n - x$ at -at most- $x + 1$ processes). Moreover, once $\mathcal{X}$ is invoked concurrently in E', it can output $n - x$ in every subsequent invocation. And (2) at the end of E, we can determine how many times an $\langle n, n - x, [1, .., 1, x + 1], [1, .., 1, x + 1]\rangle$-GSB object $\mathcal{X}$ has been invoked.[6]

Consider an $\langle n, n - x, [1, .., 1, x + 1], [1, .., 1, x + 1]\rangle$-GSB object $\mathcal{X}$ in $\mathcal{A}$. In $\mathcal{B}$, $\mathcal{X}$ is replaced with the read/write-based wait-free function $\mathsf{f}(\cdot)$ in Figure 9. $\#a_{\mathcal{X}}$ and $SP_{\mathcal{X}}[1, .., x + 1]$ are associated with the instance of $\mathsf{f}(\cdot)$ function that replaces $\mathcal{X}$. $\#a_{\mathcal{X}}$ is a constant indicating the number of times object $\mathcal{X}$ has been invoked at the end of E, hence $0 \leq \#a_{\mathcal{X}} \leq n - (x + 1)$. And $SP_{\mathcal{X}}[1, .., x + 1]$ is an $(x + 1)$-dimensional shared array, which contains a splitter object (explained below) in each of its entries.

A *splitter* is a wait-free concurrent object that provides processes with a single operation, denoted direction(), that returns a value to the invoking process. The semantics of a splitter is defined by the following properties [47, 50].

- Validity. The value returned by direction() is $right$, $down$ or $stop$.
- Solo execution. If a single process invokes direction(), only $stop$ can be returned.
- Concurrent execution. If x processes invoke direction(), then:
  - At most $x - 1$ processes obtain the value $right$,
  - At most $x - 1$ processes obtain the value $down$,
  - At most one process obtains the value $stop$.
- Termination. If a correct process invokes direction() it obtains a value.

In [47, 50] it is presented a read/write-based implementation of an splitter that has the property that if an splitter is invoked sequentially, the first invoking process gets $stop$ and all subsequent invoking processes get $right$. A

[6] We can simulate the execution E step by step and then determine the number of times an object was invoked.

slightly modification to this implementation gives an index-independent splitter in which processes call direction() with distinct inputs. Shared array $SP_{\mathcal{X}}[1, .., x+1]$ in $\mathsf{f}(\cdot)$, Figure 9, contains instances of the modified splitter algorithm in [47, 50].

Function $\mathsf{f}(\cdot)$ is simple: each p_i calls in order the splitters in SP until gets $stop$ or $right$ from an $SP[k]$; if p_i gets $stop$, then decides $\mathsf{min}(\#a_{\mathcal{X}} + k, n - x)$, otherwise it decides $n - x$.

```
% #a_X and SP_X[1, .., x + 1] are associated with
the function that replaces object X.
% #a_X (a constant) is the number of times X has been invoked
at the end of E.
% Each entry of SP_X contains an instance of the read/write-based
splitter algorithm in [50].
function f(v_i) is
(01) for k_i from 1 to x + 1 do
(02)   y_i ← SP[k_i].direction(v_i);
(03)   if ( y_i = stop)   then return(min(#a_X + k_i, n − x)) end if;
(04)   if ( y_i = down ) then return(n − x) end if
(05) end for.
```

Figure 9: Replacing an $\langle n, n-x, [1, .., 1, x+1], [1, .., 1, x+1]\rangle$-GSB object $\mathcal{X}$ in $\mathcal{A}$ (code for p_i).

Clearly, $\mathsf{f}(\cdot)$ is wait-free and index-independent. Since $0 \leq \#a_{\mathcal{X}} \leq n-(x+1)$ and due to line 03, a process decides a value in $\{1, \ldots, n-x\}$. Also, at most $x+1$ processes decide $n-x$ because at most $x+1$ processes execute $\mathsf{f}(\cdot)$. Moreover, as explained above, if $SP[k]$ is invoked sequentially, the first invoking process gets $stop$ and all subsequent invoking processes get $right$, from which follows that if $\mathsf{f}(\cdot)$ is invoked sequentially i times, $1 \leq i \leq x+1$, in the j-th, $1 \leq j \leq i$, sequential invocation, the invoking process p gets $right$ from $SP[1], \ldots, [j-1]$ and $stop$ from $SP[j]$; consequently, p decide $\#a_{\mathcal{X}} + j$, only if $\#a_{\mathcal{X}} + j \leq n - x$, otherwise it decides $n - x$. Also note that, due to the splitter specification, in every execution, at most one process can execute $SP[x+1]$, which gets $stop$ from the splitter. We now argue that at most one process decides a value $\ell \in \{1, \ldots, n-(x+1)\}$ (by the specification of $\langle n, n-x, [1, .., 1, x+1], [1, .., 1, x+1]\rangle$-GSB, at most one process can get ℓ). If two distinct processes decide ℓ, then both of them decide in line 03, and since $\#a_{\mathcal{X}}$ is a constant, we conclude that an splitter in SP outputs $stop$ at more than one process, contradicting its specification.

As already discussed, for processes in $\mathcal{B}$, each replacing function $\mathsf{f}(\cdot)$ behaves as an SQD object that solves $\langle n, n-x, [1, .., 1, x+1], [1, .., 1, x+1]\rangle$-GSB. Therefore, for any execution E' of $\mathcal{B}$ there is an execution $E'' \in S$ that is the same as E', and hence $\mathcal{B}$ read/write wait-free solves $\langle x+1, x+1, 1, 1\rangle$-GSB, contradicting [7, 45]. $\square_{Lemma\ 11}$

Lemma 12 $\forall n \geq 4 : \langle n, n+1, [1, 1, 0, .., 0], [1, 1, 1, .., 1]\rangle$-GSB $\nrightarrow_{SQD}$ $\langle n, n, 1, 1\rangle$-GSB.

Proof. The structure of the proof is similar to the proof of Lemma 11. Suppose there is an algorithm $\mathcal{A}$ that solves $\langle n, n, 1, 1\rangle$-GSB from SQD objects that solve $\langle n, n+1, [1, 1, 0, .., 0], [1, 1, 1, .., 1]\rangle$-GSB. Let S be the set containing all executions of $\mathcal{A}$ in which only the $n-2$ processes $p_3, \ldots, p_n$ participate, with identities $N-(n-1), \ldots, N$. The fact $\mathcal{A}$ is wait-free and every $\langle n, n+1, [1, 1, 0, .., 0], [1, 1, 1, .., 1]\rangle$-GSB object in $\mathcal{A}$ is SQD, imply there must exist an $E \in S$ such that (1) all processes $p_3, \ldots, p_n$ decide, (2) $p_3, \ldots, p_n$ execute $\mathcal{A}$ sequentially in some order, and (3) every $\langle n, n+1, [1, 1, 0, .., 0], [1, 1, 1, .., 1]\rangle$-GSB object $\mathcal{X}$ outputs i in its i-th (sequential) invocation, whatever the input of the invoking process; hence $\mathcal{X}$ only outputs values in the range $[1, \ldots, n-2]$.

In E, processes decide $n-2$ distinct values in the range $[1, \ldots, n]$. Let $[z_1, z_2]$ be the values that no process decides in E. Let S' be the set containing all executions of $\mathcal{A}$ that are extensions of E. Hence in every $E' \in S'$ in which both p_1, p_2 decide, they decide distinct values in $[z_1, z_2]$. Using E, we modify $\mathcal{A}$ in order to obtain a read/write-based algorithm $\mathcal{B}$ for p_1, p_2 that read/write wait-free solves $\langle 2, 2, 1, 1\rangle$-GSB, which is not possible. As in the proof of Lemma 11, processes p_1, p_2 in $\mathcal{B}$ follow the same state machine as in $\mathcal{A}$, and the initial state of the shared memory of $\mathcal{B}$ is the state of the shared memory of $\mathcal{A}$ at the end of E. Also, each $\langle n, n+1, [1, 1, 0, .., 0], [1, 1, 1, .., 1]\rangle$-GSB object $\mathcal{X}$ in p_i's code, $1 \leq i \leq 2$, is replaced with a read/write-based wait-free function $\mathsf{f}(\cdot)$. We will pick a function $\mathsf{f}(\cdot)$ that follows the behavior of SQD objects in E, namely, each such an object outputs i in i-th sequential invocation,

$1 \leq i \leq n-2$. In this way p_1, p_2 cannot distinguish they are dealing with $\mathsf{f}(\cdot)$ and not with a genuine SQD objects that solve $\langle n, n+1, [1,1,0,..,0], [1,1,1,..,1]\rangle$-GSB. In the end, each execution of $\mathcal{B}$ corresponds to an execution in S'.

Consider an SQD object $\mathcal{X}$ of $\mathcal{A}$ and let $\#a_{\mathcal{X}}$ be a constant indicating the number of times $\mathcal{X}$ has been invoke at the end of E. Recall that $0 \leq \#a_{\mathcal{X}} \leq n-2$ and $\mathcal{X}$ outputs values smaller than or equal to $\#a_{\mathcal{X}}$ in E. Let $SP_{\mathcal{X}}$ be an instance of the read/write-based splitter algorithm in [47, 50]. In the function $\mathsf{f}(\cdot)$ that replaces $\mathcal{X}$ in $\mathcal{B}$, each process p first calls $SP_{\mathcal{X}}$ and then decides as follows, according to the value, y_i, it gets from $SP_{\mathcal{X}}$: if $y_i = stop$, then p decides $\#a_{\mathcal{X}} + 1$, if $y_i = right$, then p decides $\#a_{\mathcal{X}} + 2$, if $y_i = down$, then p decides $\#a_{\mathcal{X}} + 3$.

By the specification of $SP_{\mathcal{X}}$, p_1 and p_2 cannot decides the same value in $\mathsf{f}(\cdot)$. Moreover, in sequential invocations of $\mathsf{f}(\cdot)$, the invoking process in the first invocation decides $\#a_{\mathcal{X}} + 1$, and the other one decide $\#a_{\mathcal{X}} + 2$, since, as explained in the proof of Lemma 11, if $SP_{\mathcal{X}}$ is invoked sequentially, the first invoking process gets $stop$ and all subsequent invoking processes get $right$.

The observations above imply that for any execution E' of $\mathcal{B}$ there is an execution $E'' \in S$ that is the same as E', and hence, using only read/write operations, $\mathcal{B}$ wait-free solves $\langle 2, 2, 1, 1\rangle$-GSB, contradicting [7, 45]. $\square_{Lemma\ 12}$

D An Implementation of the Splitter Abstraction

The elegant and simple algorithm described in Figure 10 implements a splitter [47, 50]. The internal state of a splitter SP is represented by two atomic multi-writer/multi-reader (nWnR) atomic registers: $LAST$ that can contain a process old name, and is initialized to any value, and a Boolean $CLOSED$ initialized to $false$.

```
operation SP.direction(v_i):
(01) LAST ← v_i;
(02) if (CLOSED)
(03)    then return(right)
(04)    else CLOSED ← true;
(05)         if (LAST = v_i)
(06)         then return(stop)
(07)         else return(down)
(08)         end if
(09) end if.
```

Figure 10: A wait-free implementation of a splitter object (code for p_i) [47, 50].

When a process p_i invokes $SP.\mathsf{direction}()$ it first writes its name in the atomic register $LAST$ (line 01). Then it checks if the "door" is open (line 02). If it has been closed by another process it returns $right$ (line 03). Otherwise, p_i closes the door, which can be closed by several processes, (line 04) and then checks if it was the last process to invoke the operation (line 05). If this is the case it returns $stop$; otherwise it returns $down$.